PATENT SPECIFICATION (11) 1367741

1367741

(21) Application No. 46571/72 (22) Filed 10 Oct. 1972

(31) Convention Application No. 188 402 (32) Filed 12 Oct. 1971 in

(33) United States of America (US)

(44) Complete Specification published 25 Sept. 1974

(51) International Classification G06F 1/00//9/00

(52) Index at acceptance G4A 4X 5B 6G 6H

(19)

(54) DATA PROCESSING SYSTEM

(71) We, NCR CORPORATION, formerly The National Cash Register Company of Dayton in the State of Ohio, and Baltimore in the State of Maryland, United States of America, a corporation organized under the laws of the State of Maryland, United States of America, do hereby declare the invention, for which we pray that a patent may be granted to us, and the method by which it is to be performed, to be particularly described in and by the following statement:—

This invention relates to data processing systems.

According to the present invention, there is provided a data processing system when conditioned by programing means which includes translation means arranged to cause the data processing system to translate a source program in a high level source language into an intermediate language which is a high level language independent of the source language, said translation means being effective, in operation, to separate executable statements (as herein defined) from non-executable information (as herein defined).

It will be appreciated that a data processing system according to the immediately preceding paragraph has the advantage that since the intermediate language is independent of the source language, the same program in different high level languages can be arranged to be translated into the same intermediate language program. This is advantageous because a single compiler can then be arranged to compile from the intermediate language into a low level language.

It should be understood herein that a "high level language" is a programming language wherein each instruction or statement generally corresponds to a plurality of machine code instructions. Examples of high level languages are ALGOL, FORTRAN, COBOL and PL/1. A "low level language" is a programming language wherein each instruction generally corresponds to a single machine code instruction. The process of converting a program from a high level language into a low level language is generally referred to as "compiling", and a program which effects such a conversion is generally referred to as a "compiler".

It should also be noted that by an "executable statement" herein is meant a program statement which specifies an operation to be performed on data. By "non-executable information" is meant information which does not specify an operation to be performed on data; for example information specifying the characteristics of data.

One embodiment of the invention will now be described by way of example with reference to the accompanying drawings, in which:—

Fig. 1 is a block diagram showing the various stages involved in a computer system using a generalized compiler;

Fig. 2 shows a block diagram of a specific embodiment wherein any of a number of commonly used source languages and any one of a class of computers are provided with language translators making them adaptable for use with the generalized compiler;

Fig. 3 is a schematic drawing of the sequential steps used in making a source program available for a specific host computer;

Fig. 4 is a schematic showing how the various manifestations of the source program are acted upon by elements of the computer system in order to transform them into advanced stages or conditions for final use of the host computer;

Fig. 5 is a schematic drawing laying out the structure of an "executable" statement in the format of a First Intermediate Language;

Figs. 6 and 7 are exemplary tree schematics illustrating information heirarchies in a new Metalanguage;

Fig. 8 is the organizational chart showing how the set of the major information structures called "groups" is subdivided into major subsets each of which are successively further subdivided into two smaller subsets;

Fig. 9 is an organization chart showing how the set of all elements of information, e, is subdivided into two major subsets and how

these smaller subsets forming a heirarchy of groups of elements of information.

Fig. 10 is an illustration of the use of the new Metalanguage to specify one type of IF statement. The information unit, Eln, is shown expanded into successively lower levels of complexity of information.

Figs. 11 through 14 are drawings of various configurations of computer systems incorporating the generalized compiler concept.

Description

The large scale integration (LSI) phenomenon in the semiconductor technology is having far reaching impact on the entire information processing industry. The availability of inexpensive and reliable hardware building blocks containing complex logic circuits opens realistic avenues for the development of more efficient computing systems. The ever increasing cost and complexity of the computer software on the other hand, necessitates a new look to the computer system architecture and hardware-software interrelationship.

Program compilation often requires more computer time than execution of the program. A significant amount of computer power is spent for the translation of high level programming languages into machine language. Compilers have increased in size from 10—40 K words for FORTRAN or COBOL up to and over 195 K words for one version of FORTRAN. Hours of expensive computer time are consumed by the compiler. Other inefficiences are caused by such tedious operations as compiler and program debugging and maintenance. In multiprogramming and time-sharing systems, compiling causes significant system overhead due to increased program swapping. Computers require larger main memories because of the core requirements of compilers.

Generally, compiler costs are an exponential function of the size of the compiler task which is incorporated into one integrated unit. Further growth in the complexity of the source language is severely limited by the increasing size and cost of the compiler using current compiler techniques. By dividing the compiler tasks into independent isolated modules which separate the user problem transformation task from the language syntax and semantics task, these limitations can be reduced and the compiling task simplified.

Compilers do not attempt to separate language translation tasks from problem transformation and consequently any change in either the source language or machine code can cause changes throughout the compiler. Source languages and computer hardware are in a constant state of flux and as a result a new compiler must be written for each source language each time a new computer is designed. Much of this work can be avoided by recognizing and taking advantage of the fact that compiler tasks can be separated into two independent classifications: language translation and problem transformation. By keeping these groups of tasks separated in the compiler, the impact of a change in the source language or a change in the computer hardware can be limited to the language translation portion of the compiler. The part of the compiler concerned with the transformation of the user's problem need not be changed because changing the language used to state the problem does not change the problem.

The generalized compiler herein is based on clearly separating "language translation" from "problem transformation". If a source program is defined as the combination of a problem algorithm and the exposition of that algorithm in some source language, then, the problem transformation consists of the mapping of the problem algorithm from a complex function-oriented format into a simpler operation sequence format. Language translation consists of changing the format of the exposition of the algorithm. To accomplish this, the source language is first translated into an intermediate language (IL), which has been designed to be independent of source language.

Then the problem, in intermediate language, is transformed from a set of procedures and algebraic expressions into a sequence of computer operations. Finally a second language transformation is performed to convert the above machine-independent output into operation codes for a specific computer. By designing the compiler as three independent modules, the impact of a change is limited. Thus, a change in source language only affects the first module, the source language translator. Likewise, the introduction of a new computer with a new machine code instruction set simply requires the change or replacement of the second language translator. In either case, the problem transformation portion of the compiler and one language translator remains unchanged. If a new computer using a new implementation language is introduced, the compiler may have to be translated into the new implementation language. Even this task is considerably smaller than translating a conventional compiler.

A large part of the compiler is in intermediate language which is independent of both the source language and the specific computer used for program execution. Consequently, it may be feasible to implement this part of the compiler in hardware since it is reasonably insensitive to changes.

Intermediate Language

In order to separate the compiler tasks into independent modules and thereby reduce the complexity, problem transformation is separated from language translation. To do this, it

is necessary to be able to state the problem in a language which is independent of the source language. This is the purpose of having an intermediate language (IL). This also defines the nature of the IL. The IL language can be divided into two independent parts: nonexecutable and executable. The nonexecutable part essentially specifies the characistics of the data and the executable part specifies the functions to be performed on the data.

All nonexecutable source program information is passed to the generalized compiler by means of tables which are collectively called a symbol table. Each identifier in the source program occupies one symbol table word. Various attributes of this identifier are coded into the proper bit positions in this word during the language translation of the source language. The "identifiers" are nounal items such as A, B, C, or X, Y, Z which identify the subject matter in a given algorithm. For example: pay=hours-worked × hourly-rate or C=A × B; Thus A, B, C, pay, hours-worked, and hourly-rate are identifiers.

The executable part specifies the functions or operations required and the sequence in which they are to be performed. The same function may be specified in a different way in each source language. However, the IL need provide only one way of specifying a specific function. The total number of functions described in IL will be greater than the number incorporated into any one source language since the various source languages do not all encompass the same set of functions. Likewise, the IL requires a larger set of operators than any single source language.

In the practical implementation herein proposed, the IL input to the generalized compiler is designed to be language independent for COBOL, FORTRAN and PL/1 thereby encompassing nearly all higher-level programs being written at present. Once this concept of an IL is established, it can be expanded by the later addition of more functions and more operators without the complications such changes would cause in current compilers. In fact, thinking of source language in terms of new problem functions may be a great aid in the more orderly development of a more powerful, simple to use, generalized source language. In addition, the separation of data attributes is compatible with the concepts of a common data base.

Source Language Translator

In order to separate the problem transformation task from the language translation, the problem must be stated in a manner that is source language independent. This is the task of the source language translator. The source language translator (SLT) strips away the excess verbiage, redundancy, and ideosyncracies from the source language. (One SLT is required for each source language.) It recognizes source language operator codes and keywords and translates these as well as executable procedures and algebraic expressions into intermediate language (IL—1 format). It places all names and declared attributes of the data and the environment into the symbol table replacing the names (operands) in the executable program with symbol table addresses. The IL—1 input format for expression may consist of alternating operands and operators. If an operand is not present between two operators in a source language expression, the SLT supplies a special "no operand" code. This is one method of eliminating a duplication of language translation within the generalized compiler.

Source languages with a precedence algorithm different from the IL precedence algorithm are translated by the addition of parentheses by the SLT.

Generalized Compiler

A Problem Transformation Module (Generalized Compiler, GC) is used to transform the executable procedures and expressions (which are in a standard IL—1 format independent of the source language) from a problem or function oriented form into a computer operation form consisting of a sequence of machine level instructions performed on elementary variables. The GC reorders the sequence of the operations of an expression according to a commonly used precedence algorithm. In addition to the expression processor routine, the GC contains routines for the transformation of all other executable statements. Appropriate code sequences are generated to transfer control to functions and sub-routines and to return as well as to test DO loop conditions and to increment to DO loop variables.

Mode Code Translator

The Machine Code Translator (MCT) is a simple language translator to translate the output operation codes of the GC into the machine codes of a specific computer. It combines symbol table information and machine code skeletons with the output (operand-operator-operand) triads of the GC.

System Organization

Fig. 1 shows the system organization whereby the Source Program (in a high-level language such as FORTRAN, COBOL, etc.) is handled by a Source Language Translator 10 (SLT) separating executable statements from non-executable information and is converted to a First Intermediate Language (IL—1). The data from the SLT 10 is then handled by the Generalized Compiler 11 (GC) which converts the statements into a Second Intermediate Language (IL—2) which is then handled by the MCT 14 in order to convert the program information into a speci-

fic Object Machine Code for the host computer.

The GC 11 provides the function of "problem transformation" with a Problem Transformation Program 12 and the function of symbol translation with Symbol Table 13.

Fig. 2 shows a more specific system where each individual high level Source Language, as C, F, or A, is handled by specific Source Language Translators 10c, 10f, 10a, etc., to convert information into IL—1. Then GC 11 provides the program information in IL—2 to specific Machine Code Translators as 14_{m1}, or 14_{m2}, or 14_{m3}, depending on the specific make of the host computer.

Source Language Translator Tasks

The four principal tasks of the Source Language Translator are:

1) Identify all source language symbols.
2) Create a symbol table of all data attributes.
3) Translate all executable source language statements into intermediate language.
4) Detect source language syntax errors.

All source language symbols must be recognized. The source language character set is translated as necessary to conform to the IL code format. Each delimiter (symbol which communicates "separation" of items of information for control purposes) is replaced by its equivalent IL code. Each identifier is added to the symbol table unless it is already listed. In either case, each identifier is replaced by its symbol table address in all executable statements.

In terms of sequences, Fig. 3 illustrates the program conditions (a), (b), (c), (d), where the Source Program, in a High Level Language, is converted to a First Intermediate Language, then to a Second Intermediate Language, and finally to being the Source Program in Object Machine Code where it is usuable for the host computer.

Fig. 4 illustrates the points of impingement of: the SLT 10 Program on the Source Program (a) to convert it to condition (b); the GC 11 program which converts the Source Program from First Intermediate Language condition (b) to Second Intermediate Language condition (c); the MCT 14 program which converts the Source Program in condition (c) from Second Intermediate Language to condition (d) into Object Machine Code.

Symbols in a Source Language may be categorized to include those shown in the following Table 1.

(A) *Delimiter*
1. Operator
   a. Primitive (+, −, *, .GE., etc.)
   b. Keyword Executable Command (D. GO TO, MOVE, etc.)
2. Punctuation (separation, control)

(B) *Identifier*
1. Keyword: Program structure name (procedure, block, division, section, etc.)
2. Statement Label
3. Variable (includes constants, literals)
4. Function
5. Subscript
6. Argument
7. Expression

(C) *Comment*

(D) *Invalid Symbol or Keyword*

All the data declaration and environment description information is placed into the Symbol Table 13. Comments are deleted. The symbol table takes the place of all nonexecutable statements when the source program is in intermediate language.

Each executable source language statement is translated into intermediate language. Essentially this consists of eliminating the redundancy and ideosyncracies and reordering the information into a language independent format. Key words are replaced by the equivalent IL code which always occurs as the first word of a statement. Arguments and parameters appear in a specified order separated by commas or some other delimiter but without the memory and readability aids found in source languages.

An "end of expression" code is added to the end of each statement function expression and the name of the expression is stored in a special area in the symbol table 13 so that the symbol table address which replaces the name in the Source Program is identifiable as an expression name. Statement functions (FORTRAN) and similar source language devices specify the solution of an equation each time the name of the expression appears.

The IL code for "NO OPERAND" is inserted in between every pair of operators in the Source Program which are not separated by an operand as necessary to avoid duplicating syntax recognition in the generalized compiler.

Each source language statement translated into intermediate language is consecutively numbered. Invalid characters are noted and converted into blanks. Error routines are invoked for invalid keywords and other syntax errors. Parentheses are added as needed to convert the precedence implied in the source language into the precedence conventions of the intermediate language in case a difference exists.

Source language syntax errors such as illegal characters, delimiters, keywords, and identifiers are detected.

Any of the currently used techniques for recognizing and translating source language can be adapted for this computer system. Syntax-directed compiling and table driven

compiling are the most common techniques used. The characteristic of the newly invented system are:

1. The translation is much simpler because it is a translation into another high level language rather than a translation into machine code.
2. The impact of a change in the source language is limited to a small part of the compiler system rather than affecting the entire computer system.

Problem Transformation Tasks

The principal tasks of the problem transformation module 12 are to:

1) Transform problem functions into computer operation sequences.
2) Optimize the object code generated.
3) Detect errors.

The executable statements are transformed from high level function-oriented format into computer hardware operation-oriented code and by rearranging the sequence in which the operations appear according to operator precedence rules of an intermediate language created as part of the system. The intermediate language operates such that:

(a) the keyword in each statement directs the problem transformation control to the routine required to transform that statement. For many statements, a direct transformation can be made using a code skeleton and substituting arguments from the statement for dummy variables.
(b) an expression processor is used to transform expressions within the statement. This processor uses one or more push-down stacks to aid in reordering expressions to meet the precedence criteria established. A push-down stack is also used to permit the use of recursion within expressions. Each time a triad (two operands and an operator) encounter an operator of lesser precedence, the triad is removed from the expression as IL output code. The triad is replaced in the expression by the symbol table address which specifies where the triad output code operation result is stored.

The object code is optimized by efforts such as the elimination of the repeated execution of the solution of the same expression whenever the values have not changed in the interim. The assignment of index registers to expedite address computation is another optimization technique. If the number of registers is limited, optimization will dictate that the innermost loops of a DO loop receive preference over outer loops. Conflicts may also occur between local and global optimization.

Error detection includes finding illegal sequences of operands and operators which the expression processor will detect. In addition, the various keyword statement routines will detect missing arguments for which no alternatives are specified and for which default values are not provided.

Second Language Translator Tasks

The principal Second Language Translator tasks are:

1) Translate the problem transformation output (triads) into machine code for a specific machine using code skeletons and symbol table information. This is accomplished by the Machine Code Translator 14 of Fig. 1.
2) Decide data type dominance for operations on operands of two different data types. Generate code needed to convert data type.
3) Assign storage locations to all object code identifiers in the symbol table. Replace the symbol table addresses of these identifiers with the assigned storage locations.

The purpose of the second language translator is to convert the machine-level machine-independent problem transformation output triads and other machine-independent instructions into instructions (primitive code) for a specific machine. The important point to keep in mind is that the only impact necessary on a compiler due to a machine code change is a language translation. Therefore, the compiler should be designed so that parts of the compiler not involved in machine code generation are independent of the features of a specific machine. This will insure that the impact of a change to a new machine will be strictly limited to the code generation skeletons. It is the job of the code generator to provide the object code necessary to execute the program instructions on a specific computer. To do this, the code generator uses the combined information of the output of the Problem Transformation Module, Expression Processor, the information in the symbol tables, and the code skeletons.

The code skeleton selected depends not only on the particular machine instruction as specified by the operator (symbols used to represent the "action" to be done on the identifiers, or operands) but also on the attributes of the data involved. The attributes of the data are specified or implied for each operand in the symbol table at the address which is the value of the operand. Each symbol table attribute field may be considered to be part of a micro-program instruction needed for a particular combination of attributes.

Every data type has a predefined dominance relationship in operations on two operands. Thus, for example, a multiply operation involving a fixed-point and a floating-point number requires machine code instructions to convert the data type of one of the numbers

before the multiply instruction is executed. In this case, floating-point dominates fixed-point and consequently, the fixed-point number is converted into floating-point. After the multiplication, the answer is converted as needed to match the type specified for the answer.

The final task of the Second Language Translator is to replace each symbol table address in the object code with the storage address allocated to it and listed in the symbol table. This task is done after the program is in the machine code for a specific target computer inasmuch as it is possible that the translation to a specific machine code may not be one-to-one. If an additional instruction were required to be inserted after storage allocation, each reference to any address beyond the point of insertion would have to be located and changed. Such a task would be far too difficult to be practical.

Description of the Intermediate Languages (IL)

Functional Organization: The function of IL is to describe the user program interfaces with the Generalized Compiler. It has formats that are independent of both the source language and the computer on which the user program is to be executed. User program statements fall into two categories:

1) Nonexecutable information
2) Executable statements

Nonexecutable statements are used for non-changing information such as data attributes, file attributes, storage allocation, and control information. In IL, this information is in the form of a symbol table.

Executable statements describe the actions to be performed on the source data during program execution in order to obtain the solution of the solution of the user's problem. Executable statements are transformed in the generalized compiler from higher level, function-oriented language format, IL—1, into machine level operation-oriented format, IL—2.

Nonexecutable Information: Before the start of problem transformation, the information from non-executable statements has been translated into symbol table entries by the SLT 10 of Fig. 1. In addition, every identifier appearing in any executable statement has been listed in the symbol table. Thus, the symbol table 13 contains the names of all identifiers in the source program and the attributes which have been specified.

Attributes are grouped into fields in the symbol table. Any attribute within a group can be specified by placing its assigned code in the field. If no code is placed into the field, a default code will be used. The default codes assigned by the compiler are loaded into the symbol table prior to the start of language translation. The programmer may declare his own choice of default attributes in his program.

In addition to data attributes, nonexecutable information in the source program includes names and sizes of arrays, relative storage locations of identifiers (COMMON and EQUIVALENCE), formats of data in external files, editing specifications, information about the hierarchial structure of data, external names, Statement Function names, entry names, function and library routines, and communications with the operating system and compiler. All of this information is placed into the symbol table 13 by the (SLT) Source Language Translator 10.

During problem transformation, additional names and facts are added to the symbol table. The names assigned to the storage of intermediate results are added. In addition, records may be kept of the most recent time at which each intermediate result has been computed in the program without a subsequent change in any of the variables used in the computation.

The symbol table is used by the (MCT) Machine Code Translator 14 module in conjunction with code skeletons to generate machine instructions. The machine language instruction sequence selected depends upon not only the operator involved but also the attributes of the operands. During this time the operands in the user's program are all symbol table addresses which makes it simple to look up the attributes.

The MCT 14 assigns storage locations to each operand in the symbol table. These addresses may be absolute or relative. Relative addressing is used, especially in multiprogramming systems, to permit the operating system to assign the actual storage to be used.

Executable Statements

At the start of problem transformation, executable statements are in IL—1 format and consist of a keyword followed by arguments each of which is followed by an EOE code (end of expression). Arguments may be elements or expressions. Expressions consist of a sequence of operand-operator pairs. Operands are symbol table addresses of either elements or expressions.

Expressions may contain expressions which contain expressions to any level of nesting within the limits of the Expression Processor push-down stack. Each expression is terminated with an EOE (end of expression) code which causes the stack to pop up one level. The EOE code of the argument is at the first level and it causes a return of control from the Expression Processor to the routine specified by the keyword of the statement being processed. The Expression Processor also returns the symbol table address of the name assigned to the expression. The name

is the address at which the value of the expression is stored. The names of expressions are stored in a special area of the symbol table so that the compiler can recognize expressions within expressions by looking at the values of the operands all of which are symbol table addresses.

During problem transformation, each executable statement is transformed into a sequence of simple instructions consisting of either an infix operator and two operands or a prefix operator and one operand.

The keyword specifies the routine to be used for the transformation. The position of the argument in the sequence of arguments establishes how it will be transformed or used by the compiler. Generally, the arguments will be processed by the Expression Processor which generates the sequence of machine instructions necessary to compute the value of the argument. The address of this value is combined with the code skeleton for the statement being processed. The code skeleton for an executable statement is a sequence of machine level instructions (triads like the output of the expression processor) with specific locations within this sequence assigned to the value of each argument. This problem transformation module contains a code skeleton for each executable statement. These code skeletons are machine-independent and are not to be confused with the code skeletons used by the code generator to generate machine code for a specific machine. Some executable statement compiler routines arrange for the assignment of index registers and analyze a larger part of the program for possible optimization. The Expression Processes optimizes at the low level of looking for duplicate triads which can be eliminated. The symbol table is used to store information needed for optimization such as the last location in a program at which a variable changed value.

Conceptually, the IL—2 code resulting from problem transformation is machine independent. The language transformation to a specific machine is accomplished by the second Language Translator, as MCT 14 of Fig. 1. The purpose of this concept is to limit the impact caused by a change in the object computer (the computer on which the program being compiled will be executed). This concept requires that the problem transformation output to the code generator be defined and designed to be set of machine operations which any of a class of machines can perform by some sequence of its own specific machine instructions or micro-program instructions.

Implementation of IL-Nonexecutable Information

The implementation of this Generalized Compiler may utilize current compiler subprograms and techniques to the extent possible without compromising this separtaion of language translation from problem transformation. The IL format for nonexecutable information is a symbol table, as 13 of Fig. 1. The symbol table can be a conventional symbol table. However, it requires more fields than the symbol table for any one language since not all languages have the same set of nonexecutable information. For example, FORTRAN does not use level numbers but Cobol and PL/1 do.

Table 2, shown below, indicates a list of the information to be incorporated into the symbol table.

This list can be reviewed and refined during writing of the compiler. It gives some perspective into the size and limits of the task. It is estimated that an average program would have five to six hundred statements and two thousand identifiers.

TABLE 2

Symbol Table Fields

1. Expression

Group	Field Contents
General	Identifier Name
	Starting Address of Expression
	component String

1. Expression (Continued)

Group	Field Contents
Type	Statement Function
	External Function
	Argument
	Triad
Compiler Notes	Last location in program at which value of expression was computed

2. Variable

Group	Field Contents
General	Identifier Name
	Level No.
	Sign
	Complement (-Unary)
	Protect Read/Write
	Link to Additional Information
	Scope of Definition in Program
Class	Scalar
	Elementary
	Label
	Group Name
	Argument
	Temporary (Compiler Use)
	Pointer (Left or Right)
	Array
	Structure

2. Variable (Continued)

Group	Field Contents
Class (Cont.)	Forward Linked List
	Double Linked List
	Tree
	Branch
	String
Type	Syntax Type Label
	Decimal/Binary
	Fixed/Floating-Point
	Real/Complex
	Arithmetic
	Logical
	Relational
	Condition
	Status
	Subscript Name
	Justified Left/Right
Storage Allocation Info.	Length (Precision)
	Length — Fractional Part
	Size in Each Dimension
	Structural Size
	Synchronized
	Contiguous to Preceding Item?
	Offset from Start of Structure
	Packed
	Filler

2. Variable (Continued)

Group	Field Contents
Storage Allocation Info. (Cont.)	Common
	Equivalence
	Initial Values
	Starting Address-Assigned Storage
	Storage Space Required
Compiler Notes	Defined at Location
	Used at Location
	Link to Next Same Type or Level
	Link to Last Same Type or Level
	Value Changed at Location

3. Linkage

Group	Field Contents
General	Identifier Name
	Relative Address in Program
	Link to Additional Arguments
	List of Dummy Arguments
	Return
Type	Entry
	External
	Function
	Sub-Routine

4. Constants, Literals, Character/Bit Strings

Group	Field Contents
General	Identifier Value
	Assigned Address
	Length
	Length-Fractional Part
Type	Constant (Numeric Literal)
	Character String
	Bit String

5. Program Structure Label

Group	Field Contents
General	Identifier Name or Number
	Program Location (Relative) Start & End
	Statement Sequence No.
	Defined
	Used
	Nesting Level
Type	Number/Name
	Statement
	Paragraph
	Section
	Division
	Do-End Group
	Begin-End Group
	Procedure

6. I/O Format

Group	Field Contents
General	Identifier Name or Number
	Assigned Address
	External Format (Picture)
	Record Size
	Block Size
	Unit Assigned
Type	Format
	Picture
	Heading
	I/O Control
	File Control
	Buffer Control
Unit	Tape Unit
	Card Reader
	Punch
	Printer
	Console
	CRT
	Disk

7. General Registers

Group	Field Contents
General	Symbolic Name
	Register Assigned
	Starting Address
	Freed at Address

7. General Registers (Continued)

Group	Field Contents
Type	Push-Down Stack
	Pointer
	Index
	Base Register

In the symbol table, the identifiers are separated into groups according to the types of information fields needed for the identifier. The groups are:

1. Expression name (Statement Function, Triad, Argument),
2. Variable name
3. Linkage—
   Entry, External, Function, Sub-routine
4. Constant, Literal, Character String, Bit String
5. Program Structure Label
6. I/O Format
7. Register Utilization

IL Implementation—Executable Statements

Basic Structure: The Source Language Translator 10 translates all executable statements of the program being compiled, into IL—1, which is the higher level language format of the input to the Problem Transformation Module. The output of the Problem Transformation Module is in a machine-level intermediate language called IL—2.

In the IL—1 format, executable statements consist of keywords and arguments. Arguments may be either simple items or expressions. Expressions are sequences of operand-operator pairs. Operands may be either simple items or expressions. Fig. 5 shows the structure of executable statements in the First Intermediate Language Format, IL—1.

The general format is: Keyword $\left\{\text{argument EOE}\right\}\ldots$

Where:

Argument	=	element expression
Element	=	symbol table address of a single item
Expression	=	$\left\{\text{operand operator}\right\}\ldots$ EOE
Operand	=	symbol table address of an element or an expression
Operator	=	code for the action to be performed on the related operands
EOE	=	code for "end of expression"

Keywords: Every executable statement in IL—1 has as its first word a keyword. The keyword specifies the location of the compiler routine used to process the arguments that follow. There is one keyword in IL—1 for each Source Program function for which compiling capability is required. Only one routine in the G. C. is required for any one function even though each source language may state the function in a different manner.

For example, the Cobol statement, "add A to B giving C" and the Fortran statement, "C=A+B" both specify the same function, namely the assignment function. Therefore, regardless of the way the function is originally stated, the same routine can process this task once it is translated into the standard G. C. input format, IL—1. Table 3 indicates the list of keywords in IL—1, the First Intermediate Language.

TABLE 3

IL Keywords (Higher Level IL)

IL Keyword	Additional source language keywords which map into the IL Keyword and its arguments.
Storage Control	
ALLOCATE	
FREE	
Operator Instructions	
DISPLAY	REPLY, EVENT, PAUSE, ACCEPT
Conditions	
ON	SNAP, SYSTEM
SIGNAL	
REVERT	
Program Control	
DELAY	WAIT
STOP	END, RUN
EXIT	
1/0 Commands	
OPEN	FILE, DIRECT, SEQUENTIAL,
	BUFFERED, UNBUFFERED, STREAM,
OPEN (Cont.)	RECORD, INPUT, OUTPUT, KEYED,
	EXCLUSIVE, BACKWARDS, TITLE,
	PRINT, LINESIZE, PAGESIZE.
CLOSE	LOCK, REWIND

TABLE 3

IL Keywords (Higher Level IL)

IL Keyword	Additional source language keywords which map into the IL Keyword and its arguments.
I/O Commands (Cont.)	
UNLOCK	
LOCATE	FILE, SET, KEYFROM, FIND, REWIND BACKSPACE.
GET	READ, FILE, INTO, SET, IGNORE KEY, KEYTO, COPY, SKIP, LIST, EVENT, NOLOCK, NAMELIST, ADVANCING, REWIND, RECORD, INVALID.
UPDATE	FILE, KEY, EVENT, DELETE, REWRITE FROM.
PUT	WRITE, FILE, PAGE, SKIP, LINE, DATA, EDIT, LIST, FROM, KEYFROM, EVENT, ADVANCING, PUNCH, PRINT.
GENERATE	INITIATE, TERMINATE, REPORT.
Problem Execution	
ASSIGNMENT	(element, array, structure,) COMPUTE, ADD, SUBT., MULT., DIV., BY NAME, ROUNDED, ON SIZE ERROR, GIVING, INTO FROM, CORRESPONDING.
DO	WHILE, TO, BY, PERFORM, UNTIL, TIMES, THRU, VARYING, FROM, AFTER, FOR,
GO TO	DEPENDING ON
ALTER	TO PROCEED TO, ASSIGN.

TABLE 3

IL Keywords (Higher Level IL)

IL Keyword	Additional source language keywords which map into the IL Keyword and its arguments.
Problem Execution (Cont.)	
IF	THEN, ELSE, IS NOT, NUMERIC, ALPHABETIC, POSITIVE, ZERO, NEGATIVE, NEXT SENTENCE
MOVE	GET, PUT, CORRESPONDING.
EXAMINE	REPLACING, TALLYING, ALL, LEADING, UNTIL, FIRST.
TRANSFORM	CHARACTERS, FROM, TO.
CONCATENATE	
SEPARATE	
CONTINUE	
CALL	TASK, EVENT, PRIORITY, ENTER, LINKAGE
RETURN	

Operators: To meet the specification that IL—1 be source language independent, the IL must describe all of the functions which can be described by any source language of practical interest. Such a task can be hopelessly difficult, resulting in an excessively large compiler if every source language feature of every language is treated as a separate function. It is important to analyze the source language features to determine what functions are actually involved.

The most elementary functions involve elementary operations to be performed on operands. These actions are specified by operators. For the IL to be language independent, the G. C. set of operators must encompass the operator sets of the various source languages of interest. From a practical standpoint, three source languages, Cobol, Fortran IV, and PL/1 have been considered in the development of the G. C. Refinements, such as the addition of a special operator from some other source language, can be easily evaluated.

In addition to the conventional operators from the above source language which specify actions on data, delimiters, needed in IL, are included. They are considered to be operators (control operators) since they also specify action to be taken.

The list of operators in the G. C. also includes three new operators created for implementation of IL. They are the prefix operators: .A., .I., and .O. . All of these operators relate to the addresses of the data involved. The first operator, .A., specifies the operation of determining the address of the associated operand value as the address of the desired data. The third operator, .O., specifies that the operand address is an offset address to which a base address is to be added.

With these new operators, it is felt that the number of source language functions which can be described clearly and simply without special routines is considerably increased. For example, PL/1 has the "Pointer" and "Offset" features which are classified as special data types. The IL address operators can clearly specify and process each of these through the Expression Processor by treating the address as a function of regular data. Likewise, list processing source languages create linked lists which can easily be translated into IL using the address operators.

Table 4 shows a list of the operators in IL and the precedence relationships. For operators in the same precedence group, the order of occurrence in an expression establishes the precedence in a first come, first executed (left to right) basis. As an exception to this rule, Precedence Level group 2, the prefix operators, are processed in a right to left order.

TABLE 4

IL Operators and Precedence

Precedence Level	IL Operators	Description
0	EOE) , .O.	CONTROL & OFFSET
1	**	EXPONENTIAL
2	+Prefix −Prefix .not. ¬ .A. .I.	PREFIX
3	* /	ARITHMETIC
4	+ −	ARITHMETIC
5	‖	CONCATENATION
6	>≥ = ≠ ≦ <	RELATIONAL
7	&	BIT STRING "AND"
8	|	BIT STRING "OR"
9	.and.	LOGICAL "AND"
10	.or.	LOGICAL "OR"
11	(FN (Function Code) SB (Subscript Code)	CONTROL

{EOE ( ) .0. , FN SB} ARE ALL CONSIDERED TO BE CONTROL OPERATORS. THEY ARE NOT A PART OF THE OUTPUT CODE.

Operands: Operands are the symbol table addresses of the identifiers in the user's program. Identifiers which are the names of expressions are kept in a separate area of the symbol table so that the Expression Processor can recognize expressions within expressions with a minimum of effort.

Expressions: Expressions in IL—1 format, the input to the Problem Transformation Module, are simply, nearly one-to-one transformations of source language expressions. Operand names are replaced by the symbol table addresses of the names. Operator codes of the source language are replaced by the operator code equivalent in IL. In order to avoid duplication of the language translation task of symbol recognition in the Problem Transformation module 12, the expression structure is designed so that every odd numbered word in any expression is an operand and every even numbered word is an operator. To accomplish this format, the source language translator adds a special code which means "no operand" whenever necessary (preceding the first operator or in between two consecutive operators).

In case the precedence of operations in a source language is different from the precedence conventions adapted for the IL, the Source Language Translator for that language must insert parentheses as necessary into the source program expressions during translation into IL unless provision is made in the G. C. to accept a user definition of the precedence rules.

Arguments: The arguments in IL—1 represent the information content of the executable statements of the user's program. Inasmuch as IL is not exposed to the programmer except during the writing of the compiler, most of the redundancy common in source languages has been removed. An argument may be a single value as a DO loop parameter or it may be an expression such as

the expression to the right of the equals sign in an assignment statement or an expression which represents the value of a parameter.

EOE: EOE is the symbolic name used to represent the internal code used in IL—1 to delimit (delimit=to separate) expressions and arguments. The code used is one of the codes not assigned for character representation in either the ASCII or EBCDIC code. This code is used to pop-up the push-down stack of the Expression Processor to the next level at the end of processing in expression and to advance the statement processing routine to the next argument at the end of compiling an argument.

Statements: Executable statements in a source language specify mappings or transformation to be performed on data during execution of the user's program. These transformations are functions which are independent of the source language is specified. The Source Language Translator 10 transforms the functional values and expressions of the source statement into the respective arguments of an IL—1 statement. The arguments in an IL—1 statement are the values or expressions to be processed by the Problem Transformation module, 12.

For example, the source language statement "GO TO 101" specifies the unconditional branch function in Fortran. In some other source language, the keyword might be "BRANCH TO" P. (or any other symbol) without affecting the function to be performed. Likewise, the identifier might be an alphanumeric name or an expression without changing the function.

Some complicated functions, which can be expressed in one sentence, are equally specified by any one of several different sets of arguments. For example, the test for continuing the iteration in a DO loop may be specified as the upper limit of the loop variable or it may be specified as a test of a logical expression in which case iteration continues for as long as the value of the logical expression is "true". It seems more practical to handle this type of situation in IL—1 by providing argument positions in the IL—1 statement for each argument of each alternative rather than imposing a more complicated language translation from source language. The Problem Transformation Module 12 recognizes each argument position by the EOE code which is present even for argument positions which have no argument. Error detecting logic is used to insure the presence of a suitable set of arguments. Tables 5(a) and 5(b) are examples of FORTRAN and COBOL source language iteration statements expressed in a newly developed Metalanguage which is described subsequently hereinafter. Table 6 shows the PL—1 source language iteration statement in the Metalanguage notation, while Table 7 shows, in Metalanguage notation, the single, generalized, First Intermediate Language (IL—1) iteration statement into which each of the Fortran, Cobol, and PL/1 statements of Table 5 and 6 map.

Tables 8(a) and 8(b) illustrate how the iteration statements from FORTRAN and COBOL map into this single IL—1 statement, in Metalanguage notation.

TABLE 5 (a)

FORTRAN Iteration Statements Written In New Metalanguage

$$a_{stmt_1}\ \text{DO}\ a_{stmt_2}\ i10 = \begin{Bmatrix} kip1 \\ ip1 \end{Bmatrix}, \begin{Bmatrix} kip2 \\ ip2 \end{Bmatrix} \begin{bmatrix} ,kip3 \\ ,ip3 \end{bmatrix}$$

—

—

$$a_{stmt_3}\ \text{DO}\ a_{stmt_4}\ i11 = \begin{Bmatrix} kip4 \\ ip4 \end{Bmatrix}, \begin{Bmatrix} kip5 \\ ip5 \end{Bmatrix} \begin{bmatrix} ,kip6 \\ ,ip6 \end{bmatrix}$$

—

—

$$a_{stmt_5}\ \text{DO}\ a_{stmt_6}\ i12 = \begin{Bmatrix} kip7 \\ ip7 \end{Bmatrix}, \begin{Bmatrix} kip8 \\ ip8 \end{Bmatrix} \begin{bmatrix} ,kip9 \\ ,ip9 \end{bmatrix}$$

—

—

—

a_{stmt_6} —

—

—

(executable program statements)

a_{stmt_4} —

—

a_{stmt_2} —

—

TABLE 5 (b)
COBOL Iteration Statements Written In New Metalanguage

PERFORM a_{proc_1} [THRU a_{proc_2}] [{kip10 / ip10} TIMES]

PERFORM a_{proc_1} [THRU a_{proc_2}] UNTIL $E1_1$

PERFORM a_{proc_1} [THRU a_{proc_2}] VARYING vn1 FROM

{kn1 / vn4} BY {kn2 / vn5} UNTIL $E1_1$] AFTER vn2 FROM {kn3 / vn6}

BY {kn4 / vn7} UNTIL $E1_2$] AFTER vn3 FROM {kn5 / vn8}

BY {kn6 / vn9} UNTIL $E1_3$]]

TABLE 6
PL—1 Iteration Statements Written In New Metalanguage

```
a1:   DO vn1 = mn1  TO mn2;
      — —
      — —
a2:   END [a1] ;
```

```
a1:   DO ;
      — — —
a2:   END [a1] ;
```

```
a1:   DO WHILE NOT E1₁ ;
      — — —
a2:   END [a1] ;
```

```
a1:   DO vn1 = mn1 [TO mn2 [BY mn3]]
      [WHILE NOT E1₁] ;
      — — —
a2:   END [a1] ;
```

TABLE 7

IL—1 Iteration Statement Written In New Metalanguage

DO c1 a1 a2 vn1 mn1 mn2 mn3 El_1

[a3 a4 vn2 mn4 mn5 mn6 El_2

[a5 a6 vn3 mn7 mn8 mn9 El_3]]

WHERE:

c1	if zero, test before iterating. If one, iterate before testing.
{a1, a2, a3, a4, a5, a6,}	are statement sequence numbers assigned by the source language translator or symbol table addresses of statement, paragraph or section labels. If a1, a3, a5 specify paragraphs or sections, a2, a4, and a6 are blank.
vn1, vn2, vn3	specify DO loop variables.
mn1, mn4, mn7	initial values of DO loop variables.
mn2, mn5, mn8	the limiting values of the DO loop variables.
mn3, mn6, mn9	the value by which the DO loop variables are incremented (or decremented, if negative) after each iteration
El_1, El_2, El_3	logical expressions which are tested before each iteration. Iteration continues until expression is true.

TABLE 8 (a)

Iteration

MAPS	
SOURCE	IL—1
FORTRAN	
a_{stmt-1}	a — i
kip1 thru kip9	mn1 thru mn9
ip1 thru ip9	mn1 thru mn9
i10	vn1
i11	vn2
i12	vn3

TABLE 8 (b)

Iteration

MAPS	
SOURCE	IL—1
COBOL	
a_{proc_1}	a1
a_{proc_2}	a2
kip10	$\begin{cases} mn1 = 1 \\ mn2 = kip10 \end{cases}$
ip10	$\begin{cases} mn1 = 1 \\ mn2 = ip10 \end{cases}$
vn4	mn1
vn5	mn3
vn6	mn4
vn7	mn6
vn8	mn7
vn9	mn9

IL—2 Format

The output of the Problem Transformation Module 12 consists of executable instructions in IL—2. Unlike IL—1, which is in higher level language, IL—2 is at the machine operation lever which means that the translation to a specific machine code from IL—2 is approximately one-to-one. The Problem Transformation Module 12 transforms each statement from IL—1 format into a sequence of instructions each of which consists of a macro call or an operation code and the one or two operands involved in this operation. Table 8(c) is an example of the IL—2 skeleton from which the IL—2 instruction sequence will be generated by the Generalized Compiler as per the sequence of tasks listed in Table 8(d) as the mapping of the IL—1 statement of Table 7.

TABLE 8 (c)

IL—2 Instruction Sequence Table for IL—1 Iteration Statement (Code Skeleton)

STMT NO	STMT ADDRESS	INSTRUCTION
1		vn1 = mn1
2		vn2 = mn4
3		vn3 = mn7
4		GO TO a7
5	a1	CONTINUE
6		IF vn1 >mn2 GO TO a2
7		IF vn1 <mn2 GO TO a2
8		IF vn1>mn2 AND mn3> 0 OR
		vn1<mn2 AND mn3<0 GO TO a2
9	a3	CONTINUE
10		IF vn2>mn5 GO TO a4
11		IF vn2<mn5 GO TO a4
12		IF vn2>mn5 AND mn6>0 OR
		vn2<mn5 AND mn6>0 GO TO a4
13	a5	CONTINUE
14		IF vn3>mn8 GO TO a6
15		IF vn3>mn8 GO TO a6
16		IF vn3>mn8 AND mn9>0 OR
		vn3<mn8 AND mn9<0 GO TO a6
17		IF $E1_1$ GO TO a2
18		IF $E1_2$ GO TO a4
19		IF $E1_3$ GO TO a6
20	a7	CONTINUE
21		<statements to be iterated from a5 to a6>
		— — —
		— — —
22		vn3 = vn3 + mn9

TABLE 8 (c) (Cont.)

STMT NO	STMT ADDRESS	INSTRUCTION
23		GO TO a5
24	a6	CONTINUE
25		<statements to be iterated from a3 to a4>
		— — —
26		vn2 = vn2 + mn6
27		GO TO a3
28	a4	CONTINUE
29		<statements to be iterated from a1 to a2>
		— — —
		— — —
30		vn1 = vn1 + mn3
31		GO TO a1
32	a2	— — —

Table 8(d) shows the tasks performed by the Generalized Compiler GC—11, with respect to mapping the IL—1 iteration statement of Table 7.

Table 8(d)
Generalized Compiler Tasks
(Statement numbers refer to statements in the IL—2 Instruction Sequence Table)

Task

1. <If a1 is blank, go to task 30 (error exit)>
2. <If a2 is blank, a1 is the symbol table address of the procedure to be iterated. Look up the a2 field in the symbol table word for this procedure. Set up the code to call this procedure (not shown in the IL—2 instruction sequence)>
3. <If a2 is given, look up in the symbol table to determine if a2 is a procedure. If it is, replace the procedure address specified as a2 with the a2 field in the symbol table word for this procedure. Set up the code to call these procedures (not shown in the IL—2 instruction sequence)>
4. <If mn1 is blank and mn2 is specified, assign a value of 1 to mn1>
5. If a3 is blank, set a3 equal to a1, a4 equal to a2, and delete statements 9, 25, 27, 28>
6. <If a5 is blank, set a5 equal to a3, a6 equal to a4, and delete statements 13, 21, 23, 24>
7. <If mn2 and mn3 are both blank, delete statements 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 22, 26, 30, 31 and go to task 29. This is a single iteration regardless of whether $E1_1$ is specified>
8. <If c1 is zero delete statements 4, 20>
9. <If $E1_1$ is blank, delete statements 17, 18, and 19 and go to task 11>
10. <If mn1 is blank, delete statements 18, 19>
11. <If mn1 is blank, delete statements 1, 2, 3, 6, 7, 8, 10, 11, 12, 14, 15, 16, 22, 26, 30, and go to task 29>
12. <If mn3 is blank, assign the value 1 to it>
13. <If mn3 is a positive literal, delete statements 7 and 8 and go to task 16>
14. <If mn3 is a negative literal, delete statements 6 and 8 and go to task 16>
15. <delete statements 6 and 7>
16. <If $E1_2$ is blank, delete statements 18, 19 and go to task 18>
17. <If mn4 is blank, delete statement 19>

18. <If mn4 is blank, delete statements 2, 3, 10, 11, 12, 14, 15, 16, 22, 26, and go to task 29>
19. <If mn6 is blank, assign the value, 1, to it>
20. <If mn6 is a positive literal, delete statements 11 and 12 and go to task 23>
21. <If mn6 is a negative literal, delete statements 10 and 12 and go to task 23>
22. <delete statements 11 and 12>
23. <If El_3 is blank, delete statement 19>
24. If mn7 is blank, delete statements 3, 14, 15, 16, 22 and go to task 29>
25. <If mn9 is blank, assign a value of 1 to it>
26. <If mn9 is a positive literal, delete statements 15 and 16 and go to task 29>
27. <If mn9 is a negative literal, delete statements 14 and 16 and go to task 29>
28. <delete statements 14 and 15>
29. <generate IL—2 output code sequence for the statements not deleted>
30. <error exit>

General Characteristics of the Intermediate Language

Table 9 lists the characteristics of the IL.

Table 9.
Characteristics of the Intermediate Language

1. It is a source language independent.
2. The input, IL—1, is function-oriented.
3. Executable input statements, in IL—1, are in higher level language format.
4. There are no data declaration statements. All data names and attributes are in a symbol table.
5. All data names not explicitly defined automatically assume default attributes.
6. All names (operands) are symbol table addresses.
7. Operator codes and punctuation codes are sufficient to describe all generally used source language operations.
8. Every executable input statement starts with a standard keyword for the function to be executed.
9. It is independent of the object computer.
10. The output, IL—2, is computer operation oriented.
11. IL—2 output code is at the machine operation level.

Tables 5, 6, 7 and 8 provide an example of the transformations involved using IL. Tables 5 and 6 show various source language statements used in FORTRAN, COBOL and PL—1 to specify the iteration function. Table 7 shows the IL—1 format which can represent any of the various source statements shown in Tables 5 and 6.

In other words, a common iterative function can be specified by a FORTRAN "DO" loop or a COBOL "PERFORM" statement or a "DO ... END" group of PL/1. Therefore, any of the above source language formats for iteration can be easily translated by SLT into a single format, IL—1, and this standardized statement can be processed by a Generalized Compiler GC—11. This Generalized Compiler GC—11 may be designed as a module such as the ALU of the host computer or it may be designed as a piece of peripheral equipment or a set of subroutines. Such a design may be economically feasible in hardware because of the wide application of the same compiler to many computers and many source languages. In this example for iteration, the intermediate language input format, IL—1, at the input to the GC—11 is shown in Table 7.

Where: The letters specify dummy variables to be replaced by scope of definition, indexing labels, index range limits, increments and conditions. The output of the GC—11 is LI—2 and consists of machine level instructions (such as: compare, increment, branch, and test) inserted in the proper sequence and locations in the text of the problem program. Although the output is at the machine code level, it is function-oriented and machine independent. The translation of this output into a specific machine code is, in most cases, a simple task of table look-up. Table 8 shows the IL—2 output instructions into which the IL—1 statement of Table 7 is transformed during problem transformation. In addition, the compiler system may add instructions to save and restore the contents of index registers as needed as other supporting operations.

Advantages of the Computer System Using Generalized Compiler Concept

The advantages of this computer system are primarily the result of organizing the work into independent decentralized functions and isolating these functions. As a result, the following advantages are obtained. These are briefly outlined first and then discussed in

Advantages:

1. Compiler design cost is less because the compiler is useful over a broader range of application resulting in economies of volume.
2. Compiler design can be optimized and refined by iteration of the results of initial design because the broader use and longer lifetime produce sufficient returns from even a small improvement.
3. Compiler is simpler to design and debug because it is organized into independent decentralized modules.
4. Compile time can be substantially reduced by implementing the design in hardware-firmware.
5. Operating system complexity can be reduced because of the reduced demands on the system by the compiler.

6. Source language development is simplified.
7. Better language features will permit more program optimization by the programmer.
8. It is compatible with the development of common data base systems and other current projects which may radically alter compiler design specifications.

Cost: The current compiler design approach requires a separate unique compiler design for each source language/machine combination. Each compiler has relatively little utilization and a short lifetime. In comparison, a large part of a generalized compiler can be used without change with numerous source languages and various makes and generations of computers. Therefore, the cost of the design is distributed over a larger number of units and is proportionately less.

Compiler Improvements: Because of the complexity of current compilers and their limited applications, once a compiler design is debugged, the design effort generally ceases. Changes are strongly resisted because of the risk of creating unsuspected new problems. In addition, it is not justified to redesign in order to overcome disadvantages discovered during operation because the potential improvement will benefit only a relatively small user group. In comparison, extensive analysis of the Problem Transformation Module of the Generalized Compiler during use is justified because it performs standard functions common to all compilers and the benefits of any improvement will be widespread.

Simplicity: The concept of separating the independent parameters of language translation and problem transformation into separate modules, in effect, decentralizes the control from one complicated control and flow sequence into three independent simple tasks. As a result, design details do not impact the whole compiler but are clearly limited. Therefore, designing and debugging are easier.

Speed: Because the Problem Transformation Module is independent of both source language and object computer, it can be used with many source languages and many computers. It will not change with changes in either source language or computer machine codes.

Therefore, it is feasible to design this unit in hardware-firmware providing a potential speed improvement of several orders of magnitude. This portion of the compiling task can be performed at nearly machine cycle speeds.

Reduced Supervision: Operating System complexity and the resulting time spent by the computer for operating system duties (nonproductive expense) are becoming major problems. As languages become extended, compilers become larger and require more operating system effort to schedule and to switch to and from active storage especially in a multi-programming environment.

In the Generalized Compiler, the language translators (SLT 10 and MCT 14) are much smaller and therefore easier to schedule and switch. The scheduling of the ALU is greatly reduced especially if the Problem Transformation Module is implemented in hardware. In addition, the switching and active storage needs are reduced. It may now be feasible to perform the entire compile task in peripheral equipment, further simplifying the operating system.

Source Language Improvements: In the Generalized Compiler, the Source Language Translator is the only module affected by a change in the source language. The translation in this module is a simple approximately one-to-one translation into intermediate language. Therefore, source language changes to better adapt the language to the user are relatively easy to implement. Likewise, the impact on the compiler is strictly confined which means that the risk of adding unsuspected errors is proportionately reduced.

Better Optimization: Small language translators, as in the Generalized Compiler, are easier to transfer into and out of active storage, and consequently source program switching from one language to another within a program may be practical. This would allow the programmer more flexibility to optimize either the writing or execution of a program. Since the programmer knows more than the compiler designer about the context of the problem being programmed, the programmer is in the best position to optimize the program globally. Local optimization, on the other hand, is probably best done by the compiler.

Reduced Obsolescence: Because the Generalized Compiler is divided into independent modules, the impact of a change is strictly limited. Likewise, it can be adapted to radical changes. For example, the work now being done on data base management and a common data base for a community of users may lead to the separation of compiler tasks into two specialized compilers, one of which compiles data attributes and environment information, and the other which compiles the executable procedures. The Generalized Compiler is not only compatible and adaptable to this concept, but may accelerate the development.

Description: New Metalanguage

From general considerations, it was felt desirable to provide:

(a) Separation of data attribute declaration from executable statements into a separate compiler function to provide a run-time symbol table or a data base so that data attributes can be taken into consideration during program execution. In other words, make provision in the compiling process to handle the common data base.

(b) Develop the Generalized Compiler concept of IL—1 so that simple translators can be designed for each source language.

(c) Develop a software writer's intermediate language, IL—2, so that operating systems and compilers can be written and debugged in a simpler fashion by using an intermediate language.

In order that this might be done, it was felt necessary to develop a new Metalanguage in order to be able to precisely define the elements of various source languages. Since a good precise definition of a "problem" is the best basis of a good "solution", it is necessary that there be provided a new Metalanguage which can be used as a tool to organize and simplify complex higher level language information. Existing metalanguages either fail to specify this information precisely or fail to be concise. If the source language statements are not specified, both precisely and concisely, the information content and, therefore, the mapping requirement will not be properly comprehended.

Thus, in order to develop, from a source language, a first intermediate language (IL—1) suitable for operation with a Generalized Compiler - - and for developing a second intermediate language (IL—2) which is suitable for easy conversion into the object machine code for any specific type of computer it was felt necessary to develop a precise method of handling and defining the information content of all the major source languages. The new metalanguage is a tool which aids in recognizing identical information among various source languages, by systematically labeling the information content of each source language statement, so that identical information units from various source languages can be mapped into the same IL—1 function.

Taking into consideration the problem of man-machine communication between source languages (such as COBOL and FORTRAN) and software writers it is seen that there is required some sort of communication link between these two elements.

Metalanguage, which is here developed as the notation used to specify elements in source languages, is the communication link between the man (the software writer) and the machine (the source language). Possibly a major cause of the chronic failure of software to be delivered on time and to perform to specifications, is the rush of software people to go to work on the final product while ignoring the importance of developing good tools for the job. Such good tools would primarily be good notation (Metalanguage) and good documentation. Many poor software developments are the direct result of the failure to properly evaluate jobs due to deficiency of notation or adequate tools for dealing with source languages.

Imprecise or poor notation, not only fails to clarify the requirements, but also creates misconceptions about the complexities and relative importance of various components of the task to be done. Thus, poor working programs often are the result of poor communication and poor communication is very often the result of poor notation (poor Metalanguage). Thus, good notation is extremely important in providing insight and good judgment for the software developer.

A Metalanguage, to be a precise and useful tool, should be designed so that it can precisely describe a language at each hierarchical level. in terms of the next lower hierarchical level. This approach is required to provide insight into the context of the problem by eliminating the saturation "noise" from lower levels in the hierarchy.

It is of great importance to recognize the information content of the statements in the source languages so that the intermediate language (IL) will preserve the information content and so that identical information content from different source languages can be recognized as such and mapped into the same intermediate language statement without redundancy.

One of the Metalanguages currently in common usage to describe computer languages is called Backus Normal Form, BNF. Although it appears to be useful for simple expressions, it is not practical to use BNF to describe complex higher level languages. In BNF, a pair of < > is used to delimit each generic term, and furthermore, a full word is generally used for each such term. This constitutes notational clutter and excess verbiage. The structure of the word used gives no indication of the amount of information involved. For example:

the notation, <integer> is "larger" than <term> although term refers to a much larger unit of information.

Thus the hierarchial levels of information cannot be easily labeled nor identified.

A precise notation can be an effective tool to aid in analysing languages to determine the actual information specified. To be useful, the Metalanguage must be capable of specifying the language-being-analyzed concisely and precisely, and it must be easy to learn and use. The Metalanguage structure itself should assist by conveying a part of the information. Thus the following requirements are stated as the specifications for a Metalanguage;

1. The meaning must be unambiguous.
2. It must be capable of describing higher level languages precisely and concisely.
3. It does not bury the major information units in a mass of detail which constitutes "noise" to the user.
4. It is capable of specifying information at each hierarchical level in terms of the next

lower level of information units.

5. It is useful as a tool to provide insight into the true nature of the problem involved.

Newly Developed Metalanguage (ML)

The Metalanguage newly developed is a set of symbols and rules designed to permit the description of statements and specifications of the syntax and the semantics rules of various higher level languages such as COBOL, FORTRAN, etc. in a precise and a concise manner. In addition, it is designed to minimize notational clutter and confusion so that it can be used as a tool in analyzing the information content of various languages. The notational structure itself conveys information about the: (a) magnitude and (b) importance, of the information being represented. The notation is designed to be easy to learn and easy to remember. Figures 8 and 9 show language concepts organized into a hierarchy of magnitude and importance with the use of special rotation to differentiate the heirarchy and meaning of concepts.

Thus, a single lower case letter (Fig. 9) is used to represent a primitive information unit or element (any one of a set of similar elements). The single lower case letter with a lower case subscript or suffix represents a subset of the set represented by the letter. Two major subsets of all information elements are "identifiers" and "operators".

It should be stressed that this described Metalanguage notation is exemplary only with respect to the symbols and scope of definitions. Other symbols and definitionscape could be used to describe the same precise method of organizing a hierarchy of concepts.

Combinations or strings of elements are specified by a single capital letter. Thus, as seen in Figures 6 and 8, the letter "E" is used to specify one member (any member) of the set of all expressions. Subsets are specified by subscripting lower case letters or numbers to the single capital letter as seen by M_1 or E_1.

Subsets are also specified by suffixing lower case letters to the single capital letter. Thus, "Eln" (Fig. 8) is any member of the subset that includes all logical numeric expressions.

In addition to the information conveyed by the structure (whether the letter is upper case or lower case) the letters assigned to various sets are selected for ease of learning and association with the set referred to. Thus, "E" is for expressions; "e" is for primitive elements; "o" is for operators, etc.

All of the above notation is for specifying "generic" sets of information. In order to specify a "particular" member of any set, the following rules apply: When a specific member is specified, it is considered to be a literal of the Metalanguage and it is written entirely in capital letters, if it is an identifier of 2 or more letters such as a reserved word. Digits and source language special symbols are written without change or extra notation. Any source language letters appearing in a literal are written as capital letters.

Since the Metalanguage symbols are not the same as those used by the source languages, there is no difficulty in distinguishing the source language symbols from Metalanguage symbols. The only exception is the three dot (. . .) repetition symbol which is a Metalanguage symbol also used as a JOVIAL language symbol. Because this symbol is so convenient and because JOVIAL is not a commonly used language, it is felt that when JOVIAL repetition notation is needed it can be specified by a sub-scripted h(h . . .).

Since the major source languages of interest do not use lower case letters, the Metalanguage lower case letters and combinations of one capital letter combined with one or more lower case letters, are easily identified as Metalanguage. Source language words of one letter or character which can be confused with the upper case Metalanguage notation (or the repetition symbol in JOVIAL) are specified by the lower case letter h with a subscript consisting of the character being specified (either a capital letter or digit or repetition symbol of or other symbol). Thus, the source language letter "A" (Metalanguage literal) is thus specified as "h_A".

One feature of BNF which has been incorporated into the new Metalanguage is the use of English language which is enclosed in corner brackets < > to describe language requirements. This permits a deferment of the design of actual code for a language until after all the specifications have been determined.

The following tables 10 through 13 show a preliminary partial assignment of the letters of the alphabet to sets and subsets and the Metalanguage symbols.

Table 10 hereinbelow shows the notation used in the Metalanguage with definitions thereof.

Table 11 provides definition of lower case letters of the alphabet and examples of use in the Metalanguage.

Table 12 (a) defines information element sets, while table 12 (b) shows the subset of various identifiers.

Table 13 defines the subsets for various operators.

TABLE 10: Metalanguage Notation

[ ]	square brackets — used to enclose an optional part of the format.
{ }	Braces — designate a choice among alternatives listed vertically. One of the alternatives must be chosen unless one of the choices is underlined designated it to be the choice by default.
——	underline — specifies default alternative.
|	vertical bar — separates alternatives. Used in simple definitions.
< >	Corner brackets — encloses English language used to describe or name a source language or intermediate language element, structure or specification.
u <i>	the number, i, inside the corner brackets specifies the number of consecutive occurrences of the unit, u.
. . .	repetition symbol — specifies that the immediately preceding unit (element, structure, or bracketed group) may occur a number of times in succession.
XX...	metalanguage literal — source language word of two or more characters with all letters capitalized, or else a source language special sumbol (character other than a letter or digit).
h_X	metalanguage literal specifying one source language character. It is specified by a lower case letter, h, subscripted by the character being specified. If the character is a letter, it must be shown as a capital letter. When no possibility of confusion or ambiguity results, the source language symbols can be written as they occur without subscripting an h. The only exception known at present is the repetition symbol (. . .) which is a legal symbol in the Jovial language. Specification of this symbol for Jovial will be by use of the subscripted h ($h_{. . .}$).
X	single capital letter — metalanguage notation for a generic term specifying a member (any member) of a set of multi-element information units such as statements (specified by the capital letter, S), and expressions (specified by the letter, E).
Xx...	capital letter followed by one or more lower case letters — specifies a multi-element member of a subset of the set specified by the capital letter. For example, Eln is the generic name for a member of the subset, Logical Numeric Expression,
x	single lower case letter — metalanguage notation for a generic term specifying a member of a set of primitive single element information units such as operators (specified by the lower case letter, o) and variables (specified by the lower case letter, v).
x_u	subscripted lower case letter — metalanguage generic term for a primitive single element which is a member of the subset $x_{u...}$ of the primitive element set specified by x. For example, o_l is the generic term for a logical operator. The set of logical operators is a subset of the set of all operators is a subset of the set of all operators, 0. Because there are so many subsets of identifiers, the major subsets have been assigned letters without subscripts in order to further reduce the clutter. Examples include i for integer, w for reserved word, v for variable. These latters are subscripted as above to form subsets.

TABLE 11: Definitions

	Used as Element	Used as Subscript	Examples
a	address label	non-numeric (alphanumeric)	m_a
		arithmetic	o_{na}
		address	o_a
b	blank	binary	v_b
c	control indetifier	condition	m_c
		complex	k_c, v_c
d	data attribute	dimension	d_d
		decimal (fixed point)	n_d, k_d
e	element	external unpacked	v_e
f	file name	flag	a_f
		prefix	o_{nf}
		floating-point	k_f, v_f
		function	o_{fn}
g	group (alternative)		
h	hollerith (literal)	hollerith	c_h, o_h
		hexadecimal	v_h, k_h
i, j	index variable (integer)	inverse (indirect address)	o_i
		fixed point	v_i
		instruction	o_{wi}
k	constant	conditional variable	v_k
l	—	logical	o_l, v_l
m	data element		
n	identifier, label, name	numeric	m_n, o_n
o	operator	optional	o_{wo}
		offset	a_o, o_o
p	parameter	pointer	a_p

TABLE 11: Definitions (continued)

	Used as Element	Used as Subscript	Examples
		picture	d_p
r		relational	o_{rel}
s	—	statement	a_s
		sign	o_{fs}
t	next instruction address	type	d_t
u		unsigned	v_u
w		reserved word	c_w, d_w
v	variable		o_w
x	unknown element		

TABLE 12

(a) Information Element Sets

e = <element>

= n | o

n = <identifier>

o = <operator>

(b) Identifier Subsets

n = <identifier-name, label, operand>

= a | c | d | m

a = = <address identifier — statement label, file>

c = <control identifier>

d = <data attribute>

m = <data identifier>

a = a_f | a_o | a_s | f | t | a_p

a_f = <flag label>

a_o = <offset address>

a_p = <memory address (pointer)>

a_s = <statement label, subroutine name>

f = <file name>

t = <next instruction address>

TABLE 12: (b) Identifier Subsets (continued)

$c = b \mid c_w$

$b = <\text{blank}>$

$c_w = <\text{reserved word used for control}>$

$d = d_d \mid d_p \mid d_w \mid d_t$

$d_d = <\text{dimension}>$

$d_p = <\text{picture}>$

$d_w = <\text{reserved word specifying attribute}>$

$d_t = <\text{type}>$

$m = m_a \mid m_n \mid v_c \mid v_l$

$m_a = <\text{non numeric data}>$

$m_n = <\text{real numeric data}>$

$v_c = <\text{complex variable}>$

$v_l = <\text{logical variable}>$

$m_a = k_{wa} \mid m_c \mid m_h \mid v_k$

$k_{wa} = <\text{non-numeric figurative constant}>$

$m_c = <\text{condition name}>$

$m_h = <\text{hollerith word}>$

$v_k = <\text{conditional variable}>$

$m_n = i \mid j \mid k_n \mid v_n$

$i = <\text{integer variable, index}>$

$j = <\text{integer variable, index}>$

$k_n = <\text{real numeric constant}>$

$v_n = <\text{real numeric variable}>$

$k_n = k_b \mid k_d \mid k_f \mid k_h \mid k_{wn}$

$k_b = <\text{binary constant}>$

$k_d = <\text{decimal (fixed point) constant}>$

$k_f = <\text{floating-point constant}>$

$k_h = <\text{hexadecimal constant}>$

$k_{wn} = <\text{numeric figurative constant}>$

TABLE 12: (b)-Identifier Subsets (continued)

v_n = $v_b\ v_d \mid v_e \mid v_f \mid v_h \mid v_l$

v_b = <binary variable>

v_d = <decimal variable>

v_e = <external unpacked numeric variable>

v_f = <floating-point variable>

v_h = <hexadecimal variable>

v_l = <fixed-point variable>

TABLE 13: Operator Subsets

o = <operator>

= $o_a \mid o_c \mid o_l \mid o_n \mid o_w$

o_a = <address operator>

o_c = <control operator>

o_l = <logical operator>

o_n = <numeric operator>

o_w = <reserved word operator>

o_a = $o_i \mid o_o \mid o_p$

o_i = <inverse address>

o_o = <offset>

o_p = <pointer>

o_c = $b \mid o_{ca} \mid o_{cn} \mid o_t$

b = <blank>

o_{ca} = <non-numeric control>

o_{cn} = <numeric control functions>

o_t = <go to next instruction>

o_l = $o_{lp} \mid o_{lr}$

o_{lp} = <logical prefix (NOT)>

o_{lr} = <logical relation (AND, OR)>

o_n = $o_{na} \mid o_{nf} \mid o_{rel}$

o_{na} = <arithmetic>

o_{nf} = <numeric prefix>

o_{rel} = <relational>

TABLE 13 (continued)

o_{nf} = $o_{fn} \mid o_{fs}$

o_{fn} = <numeric function>

o_{fs} = <sign>

o_w = $o_{wi} \mid o_{wo}$

o_{wi} = <reserved word instruction>

o_{wo} = <optional reserved word instruction>

Alternate Groupings

o_f = <prefix operator>
= $o_{fn} \mid o_{fs} \mid o_{lp}$

Operator Subsets

o_a = $o_l \mid o_o \mid o_p$

o_c = $b \mid o_{ca} \mid o_{cn} \mid o_t$

o_{ca} = <concatenation> | EOE | ,

o_{cn} = <function code> | <subscript code> | (|) | ,

o_f = $o_{fn} \mid o_{fs} \mid o_{lp}$

o_{fn} = SIN | COS | SQRT | <other functions>

o_{fs} = + | −

o_l = .I.

o_l = $o_{lp} \mid o_{lr}$

o_{lp} = NOT | .NOT.

o_{lr} = AND | OR | .AND. | .OR.

o_n = $o_{na} \mid o_{fn} \mid o_{fs} \mid o_{rel}$

o_{na} = + | − | <x> | <÷> | <exp.>

o_{nf} = $o_{fn} \mid o_{fs}$

o_o = .O.

o_p = .A.

o_{rel} = ≤ | < | = | ≠ | > | ≥ | ¬< | ¬= | ¬<

o_w = $o_{wi} \mid o_{wo}$

o_{wi} = IF | GOTO | DO | <any other reserved word instruction>

o_{wo} = WITH | THEN | AT | ADVANCING <any other optional reserved word>

o_t = <go to next instruction

TABLE 14: Information Structure Sets

(The label on the left represents a member (any member) of the set whose members meet the requirements specified on the right)

A	=	<storage structures>
C	=	<operating system routine>
D	=	<set of data attributes>
E	=	<expression — in IL—1, it consists of an alternating sequence of identifiers and operators>
F	=	<Channels, hardware interrupts, hardware indicators such as condition and overflow flags>
G	=	<Group composed of assorted structures from various subsets such as executable program statements, control instructions, expressions, elements.>
H	=	<Library>
M	=	<data structure>
N	=	<identifier dictionary (symbol table)>
O	=	<set of operators>
P	=	<procedure, program, sub-routine>
W	=	<reserved word dictionary>
S	=	<statement, including delimiter>
T	=	<decision table>
V	=	<vector — n — dimension variable>

TABLE 15: Information Structure Subsets

Expressions

E = <expression (parenthesis must be paired)>

= [of. . .] [(. . .] n [o [(. . .] [of. . .] [(. . .] n]). . .] . . .

= Ela | Eln | Era | Ern | Ea | En | Efn | Ec | El

El = Ela | Eln

Eln = <logical numeric (decimal) expression>

= {vl / Ern} [ol {vl / Ern}] . . .

Ern = <relational numeric expression>

= En [or En] . . .

En = <numeric (arithmetic) expression>

= [onf] [(. . .] mn [ona[(. . .] [onf. . .] [(. . .] mn]). . .[] . . .

Ela = <logical expression of alphanumerics>

= {vl / Era} [ol {vl / Era}]. . .

Era = <relational expressions of alphanumerics>

= Ea [or Ea] . . .

Ea = <alphanumeric expression>

= [of / oa] [(. . .] ma [{oa / oca} [(. . .] [of] [(. . .] ma]). . .] . . .

Efn = [of] ofn En

Ec = {El / n ol ncl / vk ol kn / f ol {af / mc}}

TABLE 16: Information Structure Subsets

Groups

G = <group composed of assorted structures from various subsets such as executable program statements, control instructions, expressions, elements.>

= Gt | Gf

Gt_i = <instruction to be executed if conditional expression, Ec_i, is true, If Gt_i is executed, Gf_i is by-passed>

$$= \left[Sx. \ . \ . \right]. \left\{ \begin{matrix} Sb \\ Si_{i+1} \\ ot \end{matrix} \right\}$$

Gf_i = instruction to be executed if conditional expression, Ec_i, is false.

$$= \left[Sx. \ . \ . \right] \left\{ \begin{matrix} Sb \\ Si_{i+1} \\ ot \end{matrix} \right\}$$

<the subscript, i, denotes the ith level of nested IF statements>

ot = <specifies that the next instruction to be executed during program execution is the statement which follows the entire "IF" statement including all nested "IF" statements. It is an instruction to the compiler to generate coding as needed to carry out this instruction execution sequence>

TABLE 17: Information Structure Subsets

Data Structures

M = <data structure>

m = <single identifier of data (memory)>

Ma = <array of data>

Mf = <data in a file>

Mg = <a generation of hierarchial data>

Mh = <hierarchy of data>

Ms = <a string of data>

Mt = <table of data>

TABLE 18: Information Structure Subsets

Statements		
S	=	<statement (including delimiter)>
	=	Sc | Sd | Se
Sc	=	<control instruction>
Sd	=	<data declaration statement>
Se	=	<executable statement>
	=	Sb | Sx | Si
Sb	=	<unconditional branch>
Si	=	<IF statement>
	=	IF Ec Gt owi Gf
Sx	=	<any executable statement except an unconditional branch or an IF statement>

Table 14 shows the assignment of upper case alphabet letters to the major subsets in Metalanguage. Table 15 defines the major subset "expressions" in terms of the next lower level in the Metalanguage notation.

Table 16 defines the major information subset "G" (groups) and how it is represented in Metalanguage.

Table 17 shows the assignment of suffixed capital letters to specify the various major subsets of data structures in Metalanguage notation.

Table 18 shows the assignment of labels to the major types of statements which belong to the set "S" (statements), another major information structure subset.

The Metalanguage developed herein is organized so as to define a hierarchy of concepts (useful in programming and computers) which are organized according to levels of complexity and sophistication so that there is an easy recognition of the opproximate hierarchial level of each concept.

For example, with reference to Fig. 6, certain useful information units of various complexity are shown in hierarchial relationship to each other by the line-tree drawing of Fig. 6. The P represents a program, procedure or set of programs. Now since any statement is a member of the set, S, of all statements, and since, S, is a subset of P, S, is linked to P. Other subsets of P are not shown. Now since statements contain expressions, there is a further link to E, which represents expressions showing that E, is a subset of S. Since expressions are composed of elements, there is a further link to e, which represents elements. The set of all elements breaks down into the subsets: identifiers, n; and operators, o.

The identifier set, n, may be subdivided into various subsets as shown by small letters a, b, c, f, g, h, i, d, m and the variable v, each of which can be further subdivided as, for example, the variable v in v_b, v_c, v_d, v_f, v_i, v_r.

Other subdivisions of the element set, e, might be represented by n_a, and n_a, etc.

The operator set, o, may be subdivided as seen in Fig. 6 to o_a, o_o, . . . etc.

Fig. 7 shows a further organization of hierarchies of information in the Metalanguage. In Fig. 7 the set, C, might represent the generic term for any member of the set, "control or operating system"; the letter M, might represent the generic term for any member of the set "data structure"; the letter A, may represent an area of storage; the letter P, may represent a program; the letters S, G, and T, which link to P, represents members of various sections or pieces (subsets) of the program P. Each of these symbols is further broken down into organizations involving less complex and less complicated bits of information.

Table 19 shows a COBOL "IF" statement specification which is written in IBM notation. It should be noted as a comment that some of the specifications for some of the terms in COBOL are never explicitly specified and defined in the reference manuals, as for example, the term "NEXT SENTENCE".

Table 20 illustrates the COBOL "IF" statement specification as written in Backus Normal Form (BNF).

TABLE 19

COBOL "IF" Statement Specification — Written in IBM Notation

IF arithmetic — expression —1 IS [NOT] {GREATER THAN / EQUAL TO / LESS THAN}

arithmetic — expression —2 [THEN] {statement—1. . . / NEXT SENTENCE}

{ELSE / OTHERWISE} {statement-2. . . / NEXT SENTENCE}.

Note: specifications for some of the terms are buried in the text of the manual. Other terms such as NEXT SENTENCE are never explicitly specified.

TABLE 20

COBOL "IF" Statement Specification — Written in BNF

<symbol> : := IF <arithmetic expression> IS

<symbol 1> : := <symbol> | <symbol> NOT

<symbol 2> : := <symbol 1>GREATER THAN | <symbol 1> EQUAL TO | <symbol 1> LESS THAN

<symbol 3> : := <symblo 2><arithmetic expression>

<symbol 4> : := <symbol 3> | <symbol 3>THEN

<symbol 5> : := <symbol 4><executable statement> | <symbol 4>NEXT SENTENCE

<symbol 6> : := <symbol 5> ELSE | <symbol 5> OTHERWISE

<COBOL IF> : := <symbol 6><executable statement> | <symbol 6> NEXT SENTENCE

Table 21 shows the COBOL "IF" statement specification as written in the newly developed Metalanguage.

TABLE 21

COBOL "IF" Statement In New Metalanguage

$$\text{IF Em, IS } [o_t]\ o_{rel}\ En_a\ [\text{THEN}]\ Gt\left\{\begin{matrix}\text{ELSE}\\ \text{OTHERWISE}\end{matrix}\right\}\ Gf$$

where:

$$Gt = [Sx\ldots]\ \left\{\begin{matrix}Sb\\ Si\\ o_t\end{matrix}\right\}.$$

o_t = <go to next statement following Gf>

$$Gf = [Sx\ldots]\ \left\{\begin{matrix}Sb\\ Si\\ o_t\end{matrix}\right\}.$$

o_t = <go to next statement following Gf>

Figure 10 shows generically how the "IF" statement is represented in Metalanguage. The rectangular blocks represent "strings of information units". S_i is a generic statement of "IF" and compares to the Table 21 statement.

In Fig. 10, an information unit such as E1n is connected in terms of lower order hierarchies of information as shown in the Metalanguage notation.

Use of Metalanguage Format

The new Metalanguage provides a means of specifying the information content of various source languages as well as the first Intermediate Language, IL—1, and the second Intermediate Language, IL—2, in a systematically labled common notation which defines information units in a precise, concise way. Therefore, identical information units can be easily identified. In addition, the hierarchial level of information units (degree of complexity or quantity of information) can be recognized without knowing all the elementary details of the information unit involved.

As a result of this tool (the new Metalanguage) source language statements in various languages can be written in the new Metalanguage for study and comparison to determine similarities and differences in the functions provided by the various source languages, so that the intermediate language functions can be designed in an efficient manner. Without this Metalanguage, it would be necessary to place a heavy reliance on remembering all the complications, restrictions, and specifications for the information units in each of the source languages for a statement being studied. In addition, if this statement were to be reconsidered at a later time, one would again have to refresh his memory on each of the source languages.

As a result of the work on developing intermediate languages, parameters have thus been identified and organized at various levels of an information hierarchy as follows:

(a) At least 15 major information classes have been established for large multi-element information units such as statements (S) and expressions (E).

(b) At least 35 major subsets of the above groups have been established - - as multi-element units such as logical expressions (E1).

(c) 18 principal primitive types have been established - - as sets of single elements of information such as operators (o), variables (v), etc.

(d) 44 subsets of single element groups have been established - - of which examples include types of operators such as arithmetic operators, logical operators, and control operators. Reference may be made to Figs. 6 and 7 in this regard.

One of the discoveries made in regard to the development of the information hierarchy and the new Metlanguage notation may be illustrative of the discoveries involved. In the early stages of this work, all "reserved words" (including instructions) were classified as "identifiers". Also, "instructions" were considered identifiers of call routines. After experience and work with these original classifications, it was found from experience that those "reserved words" used as instructions or commands and "instructions" should be designated and defined as "operators" rather than as identifiers.

Hereinbelow will be found a group of tables which illustrate how various statements from major source languages can be written in the new Metalanguage notation and how they transform into statements in the first and second intermediate languages and how they may be mapped.

The following tables, tables 15—24 will show the representation of various COBOL and FORTRAN functions in Metalanguage notation:

Table 15 shows representative functions which are found in high level languages in the left column, while the equivalent function in IL—1 format is shown in the right hand column.

TABLE 15

Representative Functions Found In Programs	Representation in IL—1 Format Using Metalanguage Notation
Logival IF	IF El Gt Gf
Relational	IF El Gt Gf
Sign (COBOL)	IF El Gt Gf
Sign (FORTRAN)	IFS En S_1 S_2 S_3
Class	IFCL n ncl Gt Gf
Condition	IFC f vk g Gt Gf
On Count	ON K_{11} K_{12} K_{13} Gt Gf
Unconditional Branch	BR as
Alter (assigned Branch)	ALTER as-i as
	GO as, as1, as2, . . .
Computed Branch	GOTO i, as1, as2, . . . asj
Iteration	DO cl al a2 vn1 mn1 mn2 mn3 El_1 [a3 a4 vn2 mn4 mn5 mn6 El_2[a5 a6 vn3 mn7 mn8 mn9 El_3]]
Assignment	ASGNMT ow crd M E Se mn4

Table 16 shows the Conditional Branch (IF) of source language FORTRAN and source language COBOL in Metalanguage notation. Table 16B for COBOL shows subportions (a), (b), (c) showing, in Metalanguage notation, three forms of conditional branch (IF) tests.

Table 16 subsection (d) shows the single IL—1 statement into which everyone of these FORTRAN and COBOL statements map.

Subsection (e) lists the Generalized Compiler tasks for these conditional branch (IF) statements.

Subsection (f) of Table 16 shows a typical portion of the conditional branch IF statement in IL—2.

Subsection (g) illustrates how the conditional branch IF statement is mapped.

TABLE 16: Conditional Branch (IF)

(compared after alignment of decimal points if all elements numeric)

(A) FORTRAN

IF (Eln) St

(B) COBOL

(a) LOGICAL TEST

IF El [THEN] Gt {ELSE / OTHERWISE} Gf at

(b) RELATIONAL TEST

IF E_1 IS [NOT] { > / = / < / GREATER THAN / EQUAL TO / LESS THAN } E_2 [THEN Gt {ELSE / OTHERWISE} Gf at

(c) SIGN TEST

IF En IS [NOT] {POSITIVE / ZERO / NEGATIVE} [THEN] Gt {ELSE / OTHERWISE} Gf at

(d) IL—1

IF_1 El_1 Gt_1 Gf_1

at

|

|

|

(e) G. C. TASKS

<symbol table look-up of attributes of all elements>

<code to align decimal points if attributes of all elements are numeric>

<rearrange expression according to precedence assigned to operators>

<generate code to compute the value of the expression>

<code to test logical value>

<code to execute Gt and skip Gf if test result true>

TABLE 16: Conditional Branch (IF) Continued

<code to execute Gf)>

<repeat above steps for each IF statement>

<repeat .— —>

<the next statement to be executed > ot < is the statement following the last nested "IF" statement. In the IL—1 format it is at > at. <Nested "IF" statements can appear in Gt or Gf>

(f) IL—2

a o n [n]

(g) Conditional Branch (IF)

MAP

SOURCE	MAPS INTO
st	= Gt
Gt_i	= [Sx. . .] {Sb, Si_{i+1}, ot}
Gf_i	= [Sx. . .] {Sb, Si_{i+1}, ot}
NEXT SENTENCE	= GO TO t
{<executable statement . . . following THEN or El and delimited by ELSE or OTHERWISE or a keyword designating a new instruction>}	= Gt
{<executable statement . . . following ELSE or OTHERWISE delimited by the start of a new instruction>}	= Gf
E_1 orel E_2	= El
En orel O.O	= El

TABLE 16: (g) Conditional Branch (IF) Continued

SOURCE		MAPS INTO (IN IL—1)
GREATER THAN |	>	= orel
EQUAL TO |	=	
LESS THAN |	<	
NOT GREATER THAN |	¬>	
NOT EQUAL TO |	¬=	
NOT LESS THAN |	¬<	
POSITIVE |	$>_0$	= orel 0.0
ZERO |	$=_0$	
NEGATIVE |	$<_0$	
NOT POSITIVE |	$¬>_0$	
NOT ZERO |	$¬=_0$	
NOT NEGATIVE |	$¬<_0$	

TABLE 17 shows the FORTRAN Sign Test stated in Metalanguage (a) and the equivalent statements in IL—1 (b) and in IL—2 (c). Subsection (d) of Table 18 shows the mapping.

TABLE 17: Sign Test

(a) FORTRAN

IF (En) S_1, S_2, S_3

(b) IL—1

IFS En, S_1, S_2, S_3

(c) IL—2

<check symbol table — all elements of E must be numeric>

<code to compute value of En>

COMP En 0.0

BRL S_1

BRE S_2

BRG S_3

(d) MAP

S_1 = <next statement to be executed if the value of the relational expression is negative.>

S_2 = <next statement to be executed if the value of the relational expression is zero.>

S_3 = <next statement to be executed if the value of the relational expression is positive.>

Similarly to Table 17, table 18 shows the COBOL Class Test; Table 19 shows the Condition Test for COBOL; Table 20 shows the COBOL function of Conditional Branch on Count; Table 21 shows the COBOL and FORTRAN Unconditional Branch functions; Table 22 illustrates the COBOL and FORTRAN Alter functions.

TABLE 18: Class Test

COBOL

IF n IS [NOT] { NUMERIC / ALPHABETIC } [THEN] Gt { ELSE / OTHERWISE } Gf

IL—1

IFCL n, ncl, Gt, Gf

IL—2

<code for character by character table look-up of class specified by ncl>

BRE Gt

EXEC Gf

TABLE 19: Condition Test

COBOL

IF vk IS [NOT] kn [THEN] Gt { ELSE / OTHERWISE } Gf.

IF [NOT] { af / mc } [THEN] Gt { ELSE / OTHERWISE } Gf.

IL—1

IFC f, vk, g, Gt, Gf

IL—2

<look up in symbol table to determine if mc or af or kn>

<code to find mc corresponding to kn if vk specified>

<code to address and fetch the conditional variable field vk in the record currently in the file location,. f>

<compare> vk g

BRE Gt

EXEC Gf

TABLE 19: Condition Test Continued

MAPS

where: g = mc | af | kn

kn = <program supplied test value specifying one of the condition names which the conditional variable may represent>

vk = <conditional variable (source program translator generates name and assigns a value of 1 to it in the case of forms overflow test)>

<translator supplies the file name, f, of the file currently open.>

TABLE 20: Conditional Branch On Count

COBOL

ON ki1 [AND EVERY ki2] [UNTIL ki3]

Gt[{ELSE / OTHERWISE} Gf]

IL—1

ON ki1 ki2 ki3 Gt Gf

COMPILER TASKS

<assign name to counter variable, i. add name to symbol table and set its value to zero>

<default ki3 — infinity. For both ki2 and ki3 blank (execute Gt once)

default ki2 = ki1, ki3 = ki1 + 1>

IL—2

i = i + 1

IF i≥ki3 GO TO a4

a1 IF (ki1 — i) a2, a3, a4

a2 ki1 = ki1 + ki2

IF (ki1 — ki3) a1, a4, a4

a3 Gt

a4 G

a4 Gf

TABLE 21: Unconditional Branch

FORTRAN

GO TO a_{stmt}

COBOL

GO TO $\left\{\begin{matrix} a_{para} \\ a_{sec} \end{matrix}\right\}$

IL—1

BR as

IL—2

BR as

MAPPING REQUIREMENTS

SOURCE		IL—1
$\left\{\begin{matrix} a_{stmt} \\ a_{para} \\ a_{sec} \end{matrix}\right\}$	=	as

TABLE 22: Alter

(a) FORTRAN

ASSIGN a_{stmt_1} TO a_{stmt}

GO TO ($a_{stmt\ 1}'$ $a_{stmt\ 2}'$. . .)

—

(b) COBOL

ALTER $\left\{a_{para\_1}$ TO PROCEED TO $\left\{\begin{matrix} a_{para} \\ a_{sec} \end{matrix}\right\}\right\}$;. . .

—

—

—

GO TO [a_{para}].

TABLE 22: Alter. Continued

IL—1

{ALTER as-i, as} . . .

—

—

{GO as as1 as2} . . .

IL—2

<check that as-i in the ALTER statement is a legal member of the set in the GO statement.>

<look up in symbol table the address (loc2) of the branch to as>

<code to move instruction at location 1 to location 2>

<code to branch past location 1>

loc 1: <code to branch to as-i>

—

—

—

loc 2: <code to branch to as>

EXAMPLE:

```
MVAR $1, $2, 4
BR  $1 + 4
$1 BR as-i
      —
      —
      —
$2 BR as
```

MAP

SOURCE	IL—1
a_{stmt-i}	as-i
a_{stmt}	as
a_{para_i}	as-i
a_{sec}	as
a_{para}	as

If a_{para} is omitted in the COBOL statement, the GO TO statement must have a paragraph name, by the only statement in the paragraph and be modified by an ALTER statement before 1st execution.

Table 23 illustrates the Computed Branch functions for FORTRAN and COBOL, which, as the above, includes mapping requirements. Table 24 shows some of the Assignment Statement Varieties in FORTRAN (A) and COBOL (B), followed in sequence the statements in IL—1 format (c), the Generalized Compiler tasks (d), the IL—2 format (e), and the mapping (g).

Subsection (h) of Table 24 shows mapping details applicable to specific COBOL source statements corresponding to subsection B as related to IL—1 format.

TABLE 23: Computed Branch

FORTRAN

GO TO (a_{stmt_1}, a_{stmt_2}, . . .), i

COBOL

GO TO $\left\{ \begin{matrix} a_{para_1} \\ a_{sec_1} \end{matrix} \right\} \left[\begin{matrix} ,a_{para_2} \\ ,a_{sec_2} \end{matrix} \right] \ldots$ DEPENDING [ON] i

IL—1

GO TO i, as1, as2, . . .asj

IL—2

	BR 1 + *
<i = 1>	BR as1
<i = 2>	BR as2
. . .	|
	|
	|
<i = j>	BR asj

Mapping Requirements

In COBOL, if i>j, the GO TO statement is ignored and control passes to next sequential statement.
* designates current instruction address.

SOURCE	IL—1
a_{stmt_i}	as-i
a_{para_i}	as-i
a_{sec_i}	as-i

TABLE 24: Assignment Statement

(A) FORTRAN

(1) vl = El

(2) vn = En

(B) COBOL

(3) COMPUTE $\left\{\begin{matrix} mr \\ vn \end{matrix}\right\}$ [ROUNDED]

$= \left\{\begin{matrix} En \\ kn \end{matrix}\right\}$ [; [ON] SIZE ERROR Se . . .]

(4) ADD mn2 . . . TO mn1 [ROUNDED]

[; [ON] SIZE ERROR Se . .]

(5) ADD mn2 mn2 . . . GIVING mn1

[ROUNDED] [; [ON] SIZE ERROR Se . . .]

(6) ADD CORRESPONDING Mn2 TO Mn1

[ROUNDED] [; [ON] SIZE ERROR Se . . .]

(7) SUBTRACT CORRESPONDING Mn2

FROM Mn1 [ROUNDED]

[r [ON] SIZE ERROR Se . . .]

(8) SUBTRACT mn2 . . .

FROM $\left\{\begin{matrix} vn1 \text{ [GIVING vn1]} \\ kn \text{ GIVING vn} \end{matrix}\right\}$

[ROUNDED] [; [ON] SIZE ERROR Se . . .]

(9) SUBTRACT mn2 FROM mn1

[ROUNDED] [; [ON] SIZE ERROR Se . . .]

TABLE 24: Assignment Statement Continued

(B) COBOL

(10) MULTIPLY mn2 BY { vn1 [Giving vn] / kn GIVING vn }

[ROUNDED] [; [ON] SIZE ERROR Se . . .]

(11) DIVIDE mn2 INTO { vn1 [GIVING vn] / kn GIVING vn }

[ROUNDED] [REMAINDER mn4]

[; [ON] SIZE ERROR Se . . .]

(12) DIVIDE mn3 BY mn2 GIVING vn

[ROUNDED] [REMAINDER mn4]

[; [ON] SIZE ERROR Se . . .]

IL—1

ASGNMNT ow crd M E Se mn4

G. C. TASKS

<process expression, E, to transform it from algebraic notation into an operation sequence based on standard precedence rules>
<the symbol table is provided with the updated length of all expressions, E, and executable statement groups, Se>
<if the source language precedence rules are not standard, the source language translator must add parentheses as needed to convey the correct precedence information>

IL—2

a [n] o n

— — —

— — —

MAPPING

TABLE 24: Assignment Statement Continued

GENERAL SOURCE LANGUAGE	IL—1
vl	M
vn	M
mr	M
vnl <with vn blank>	M
mnl	M
=El	E
=En	E
=kn	E
ROUNDED	set crd =1
ON SIZE ERROR Se . . .	Se
element expression	ow = 0
ADD CORRES	ow = 1
SUBT. CORRES	ow = 2
CONCATENATE	ow = 3
SEPARATE	ow = 4
Mn_1 <=mnl-1 . . .>	M
Mn_2 <= mn2-i . . .>	E<E_1 [o E_i] . . .>
SOURCE STATEMENT NO.	
(4) mn2[+ mn2] . . . +mnl	E
(5) mn2{+ mn2} . . .	E
(8) $\left\{\begin{matrix}vnl\\kn\end{matrix}\right\}$ {-mn2} . . .	E
(9) mnl — mn2	E
(10) mn2 * $\left\{\begin{matrix}vnl\\kn\end{matrix}\right\}$	E
(11) $\left\{\begin{matrix}vnl\\kn\end{matrix}\right\}$ / mn2	E
(12) mn3 / mn2	E

Operation

In the preceding sections involving description, there has been presented a computer system which is enhanced by an optimal use of a generalized compiler. Fig. 1, shows a flow diagram whereby the source program in high level language is communicated to a source language translator, SLT 10, wherein there is a separation of non-executable information and executable statements (in the IL—1 language format). Then the non-executable information is mapped into the Symbol Table 13 of the Generalized Compiler 11, and the executable statements are communicated to Problem Transformation Module 12. Subsequently, this information is communicated from the Generalized Compiler 11 to the Machine Code Translator MCT 14 which provides an output of the source program in primitive operation sequence code.

Fig. 2 illustrates a more specific embodiment showing how each high level source language (such as COBOL, Fortran, etc., is provided with its own specific source language translator (SLT) respectively, such as 10_c, 10_i, and 10_a. Further, Fig. 2 also shows that each host computer, such as No. 1, No. 2, and No. 3 is provided with its own individualized program package (MCT) designated as 14_{m1}, 14_{m2}, and 14_{m3}.

Figures 11 through 14 show various configurations of the system which can be used to suit particular requirements.

Fig. 11 shows a computer system configuration using the GC for the conventional, current processing sequence of compiling a source program into an object program in machine code for a specific current type machine, the object program having been processed to the extent that it is ready for the computer to link, load, and execute the program to process the data input supplied at run-time. When said object program is not being used to control the processing of data, it is stored on cards, tape, disc files, or other secondary storage media. In Fig. 11 a sequential system of program operations is shown wherein the source program 30 is operated on by the Source Language Translator 10. The program, in IL—1 format, is then operated upon by the Generalized Compiler 11 having an output program in IL—2 format. This program is processed by MCT 14 to provide an object program 42 which is in machine code for a specific host computer. Data, as represented in block 35, is combined for operation with the object program 42, this object program being the source program stored on cards, tape, or disc at times occurring between executions of the object program. Following this is the run-time machine code execution routines designated as 44 which provides the output designated as block 45.

This configuration replaces the standard or conventional compiler functions where great amounts of memory and computer time are used to accomplish compilation by means of a separate compiler for each source language.

In Fig. 11 block designated as the object program 42 is the output of the compiler which can be stored on cards, tape, or disc with no necessity for actual execution of the program. In other words, it may be held and stored for later execution as in conventional, current computer systems. Line 39 indicates that, subsequently, at some convenient time, the object program 42 may be utilized for execution in combination with the data block 35.

Fig. 12 shows a computer system configuration using the Generalized Compiler in which the source program is processed into an object program and a symbol table at the IL—2 level. These are stored in secondary storage until the program is to be executed. In this configuration, the IL—2 is the implementation language of the target or host computer which means that the computer has run-time interpretive routines which can execute the object program directly. Run-time interpretive routines perform the final translation of IL—2 into primitive machine codes as necessary, and provide for type conversion (for instance, integer to floating point) as necessary based on data attributes furnished in a symbol table, or as part of the input data, or in a data base.

Object programs which are in IL—2 language are less vulnerable to obsolescence or change than a conventional object program written in machine code for a specific machine. Object programs in IL—2 can be used on any of a class of machines without recompiling the source programs. Recompiling would be necessary in the event the new machine used a different implementation language. However, the implementation language does not change as often as machine codes since the implementation language, being IL—2, is designed to be machine-independent for a class of machine.

In Fig. 12 source program 30 is processed by SLT 10 and GC 11 into an object program in IL—2 designated block 42. Thus, this object program in IL—2 can later be executed in any one of a class of computers. Using Run Time Interpretive routines 43 to process the block 35 as per the object program instruction sequence and the Symbol Table 13' (which may be in a data base), an Output 45 of processed data may be efficiently realized.

Figs. 13 and 14 illustrate computer system configurations which differ from each other only in that in Fig. 14, a data base 36 is used rather than a Symbol Table as in Fig. 13. These particular computer systems (Fig. 13 and Fig. 14) illustrate an approach to computer architecture that is not feasible without

the Generalized Compiler concept. This approach visualizes the general use of large common data bases and the elimination of data declaration from a large class of source programs thereby considerably simplifying the writing of source programs. In conjunction with the data base, a separate compiler can be used to update the data declaration information as needed.

Because the Generalized Compiler is designed to be used with various source languages making it a frequently used part of the computer system and because of the advent of LSI technology reducing the cost of hardware-firmware, it is feasible to consider implementing the various subroutines of the Generalized Compiler on LSI chips. This use of high speed circuitry could increase the compile speed by an order of magnitude thereby doubling the total throughput of a computer system.

In the configuration of Figs. 13 and 14, the source program is translated by SLT 10 (which might also be implemented in hardware-firmware) into an object program 41 in IL—1 and a Symbol Table 13 (unless a data base 36 is being used). The object program 41 and Symbol Table 13 are stored in secondary storage until execution of the program is desired. During program run-time, the Generalized Compiler 11 maps the source program statements from IL—1 into IL—2, mapping as much as one block or procedure at a time. The run-time interpretive routines 44 process or execute these IL—2 commands in conjunction with the input data 35 and data attribute information from the data base 36 or symbol table 13 producing program output 44'. In case of programs which are to be run many times, the output of the Generalized Compiler in IL—2, microcode, or other primitives, may be stored on secondary storage as a new object program 45' which will be used in subsequent executions of the program bypassing the Generalized Compiler.

The Generalized Compiler—Summary

As a result of the language development and use to date, it is now possible to specify, with reasonable assurance, a basic general set of functions which can be used to describe the great majority of user programs being written today regardless of the source language in which written. This makes possible a new simplifying approach to compiler design.

Instead of trying to fit the compiling task into one elegant overall concept such as a Syntax-directed compiler or a compiler-compiler, the system described herein has divided the task according to independent parameters. Thus, language translation is separated from problem transformation; in other words, how the problem is stated is separated from the problem itself. In addition, the description of the data attributes and environment (data declaration) is separated from the actions to be performed on the data (the executable statements).

As a result, compiler complexity has been reduced without placing new restrictions on the user. Instead of designing a completely new compiler for each language for each new computer, all that is required is a new source language translator to translate from one high level language to another when a change in source language is involved and a new translator to translate from one machine-level code to another when a new computer is involved.

To implement this concept of separating problem transformation from language translation. Two intermediate languages IL—1 and IL—2, were developed. Nonexecutable information in IL is in the form of a symbol table. The executable input to the Problem Transformation Module is in high level language, IL—1, which is source language independent. The executable output from the Problem Transformation Module is in machine-level operation code format, IL—2, which is machine independent for a class of machines.

As a result, the Problem Transformation Module need not be redesigned for either a source-language or a computer change. Because of this design stability, the Problem Transformation Module can be designed as a hardware-firmware unit thus reducing the work load of the CPU and the operating system and increasing the speed of compiling.

As a further exciting possibility, the flow of jobs through a computer may be radically changed so that data attributes are stored as part of a common data base and are compiled and updated independently from executable programs by a specialized compiler which only compiles data attributes and environment information, assigns storage addresses to data, and otherwise interfaces with the common data base. Users using common data already in the common data base do not have to declare the attributes. Mixed data types are converted as necessary as they are fetched from the common data base during run time.

As another possibility, the source language can be some conversational language which can be compiled on-line assuming the conversational language is a simple one-to-one translation into IL—1 and assuming that the Problem Transformation Module is a hardware-firmware unit, especially if the source syntax recognizer is also implemented as firmware.

In summation, the compiler system described herein promises to make the computer function more manageable and thereby

to allow more freedom and flexibility to develop better computer systems and user languages.

WHAT WE CLAIM IS:—

1. Data processing system when conditioned by programming means which includes translation means arranged to cause the data processing system to translate a source program in a high level source language into an intermediate language which is a high level language independent of the source language, said translation means being effective, in operation, to separate executable statements (as hereinbefore defined) from non-executable information (as herein defined).

2. Data processing system according to Claim 1, wherein said non-executable information is stored in tabular form in a symbol table.

3. Data processing system according to Claim 2, wherein operands in said intermediate language are formed by symbol table addresses.

4. Data processing system according to any one of the preceding claims, wherein said programming means includes compiling means arranged, in operation, to cause the data processing system to convert said source program from said intermediate language into a second intermediate language which is a low level language.

5. Data processing system according to Claim 4, including means for storing said source program in said second intermediate language.

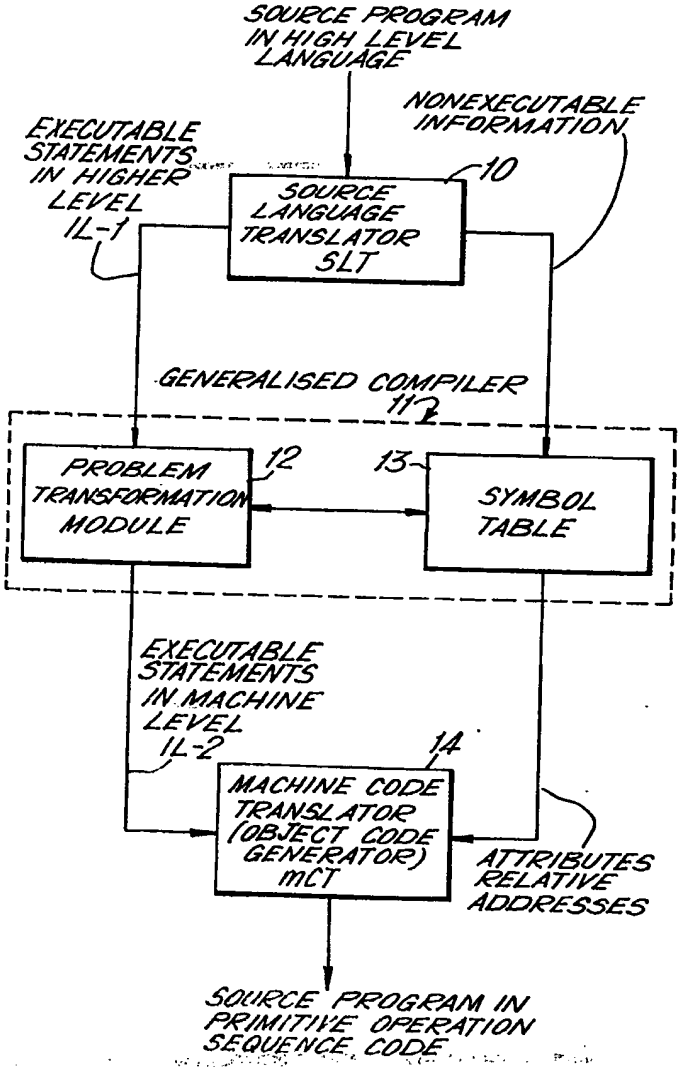
6. Data processing system according to Claims 4 or 5, wherein said programming means includes second translation means arranged, in operation, to cause the data processing system to translate said source program from said second intermediate language into a machine language.

7. Data processing system substantially as hereinbefore described with reference to the accompanying drawings.

R. G. ROBINSON,
Chartered Patent Agent,
Agent for the Applications.

Printed for Her Majesty's Stationery Office by the Courier Press, Leamington Spa, 1974.
Published by the Patent Office, 25 Southampton Buildings, London, WC2A 1AY, from which copies may be obtained.

Fig. 1. OVERALL FLOW DIAGRAM COMPUTER SYSTEM USING GENERALISED COMPILER.



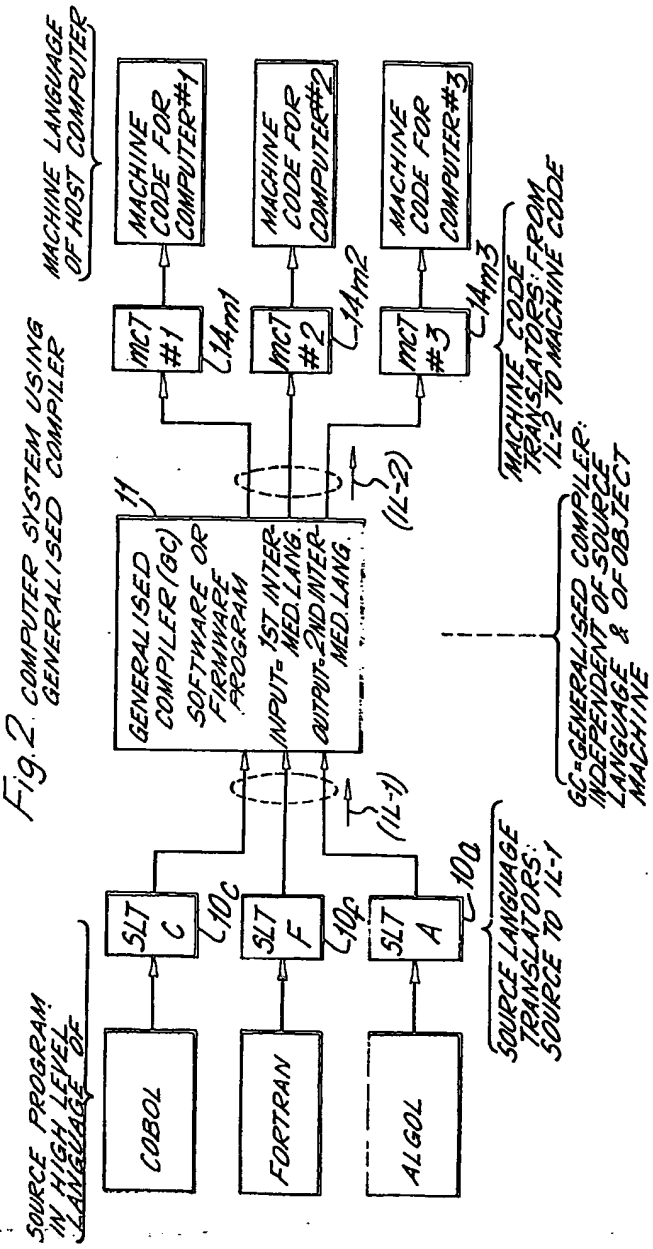


Fig. 2 COMPUTER SYSTEM USING GENERALISED COMPILER

Fig. 3.

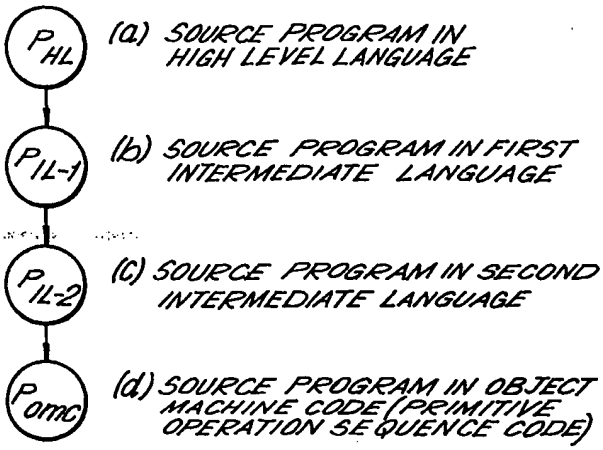
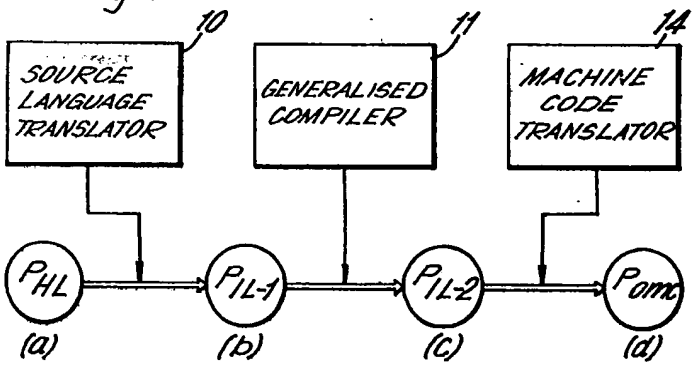

P_{HL}
(a) SOURCE PROGRAM IN HIGH LEVEL LANGUAGE
P_{IL-1}
(b) SOURCE PROGRAM IN FIRST INTERMEDIATE LANGUAGE
P_{IL-2}
(c) SOURCE PROGRAM IN SECOND INTERMEDIATE LANGUAGE
P_{omc}
(d) SOURCE PROGRAM IN OBJECT MACHINE CODE (PRIMITIVE OPERATION SEQUENCE CODE)

Fig. 4.


10
SOURCE LANGUAGE TRANSLATOR
11
GENERALISED COMPILER
14
MACHINE CODE TRANSLATOR
P_{HL}
P_{IL-1}
P_{IL-2}
P_{omc}
(a)
(b)
(c)
(d)

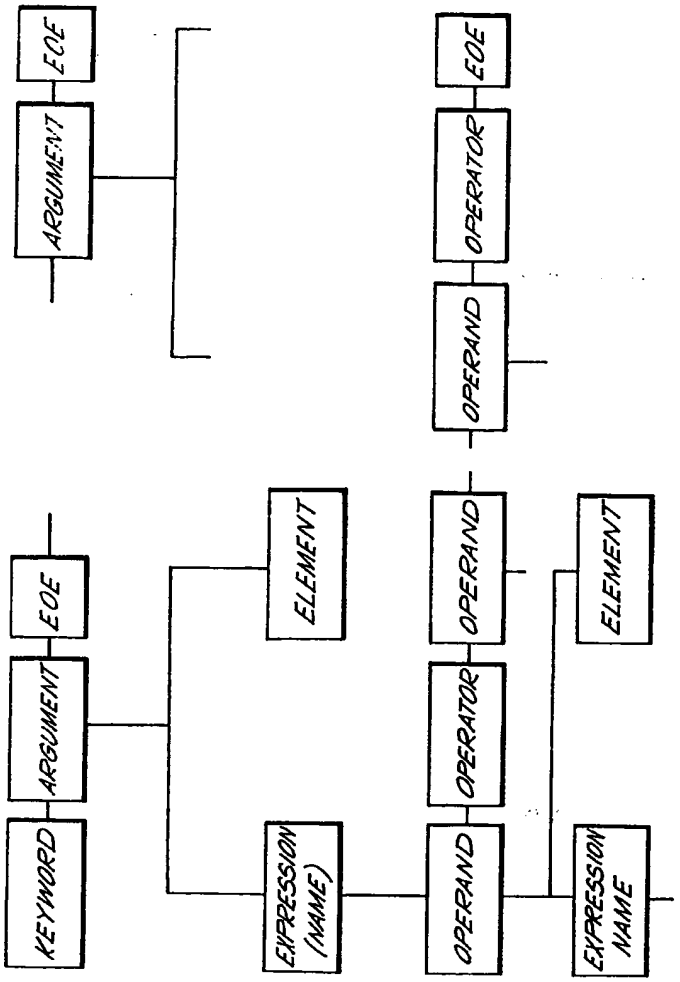

EOE
ARGUMENT
EOE
OPERATOR
OPERAND
EOE
ARGUMENT
KEYWORD
ELEMENT
EXPRESSION (NAME)
OPERAND
OPERATOR
OPERAND
ELEMENT
EXPRESSION NAME

Fig. 5.

Fig.6. METALANGUAGE-EXEMPLARY INFORMATION ELEMENTS (SETS)

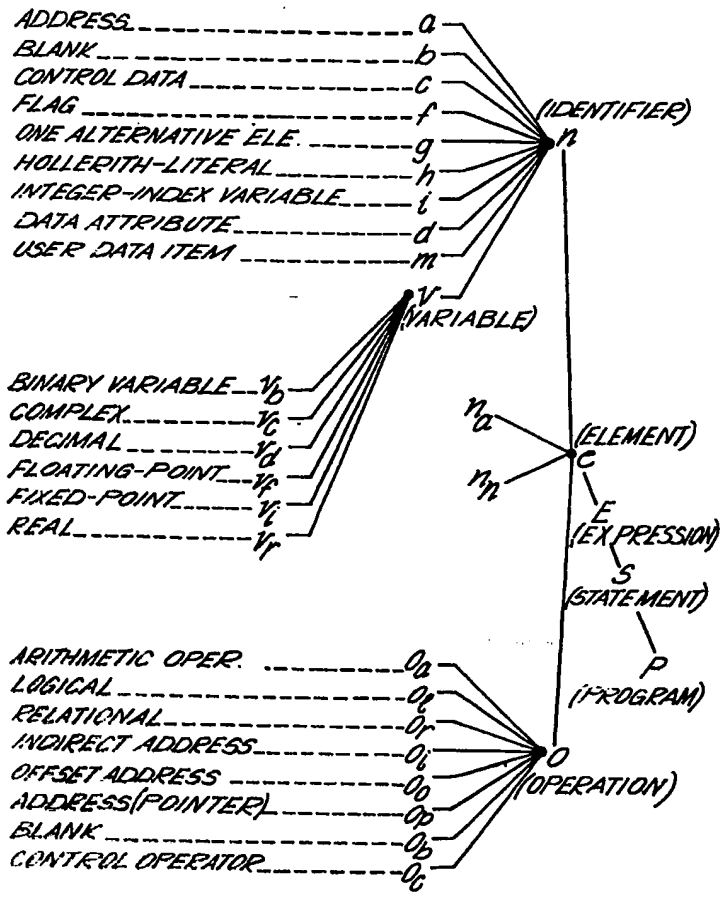


Fig. 7. EXEMPLARY HIERARCHICAL TREE OF NOTATIONS IN METALANGUAGE

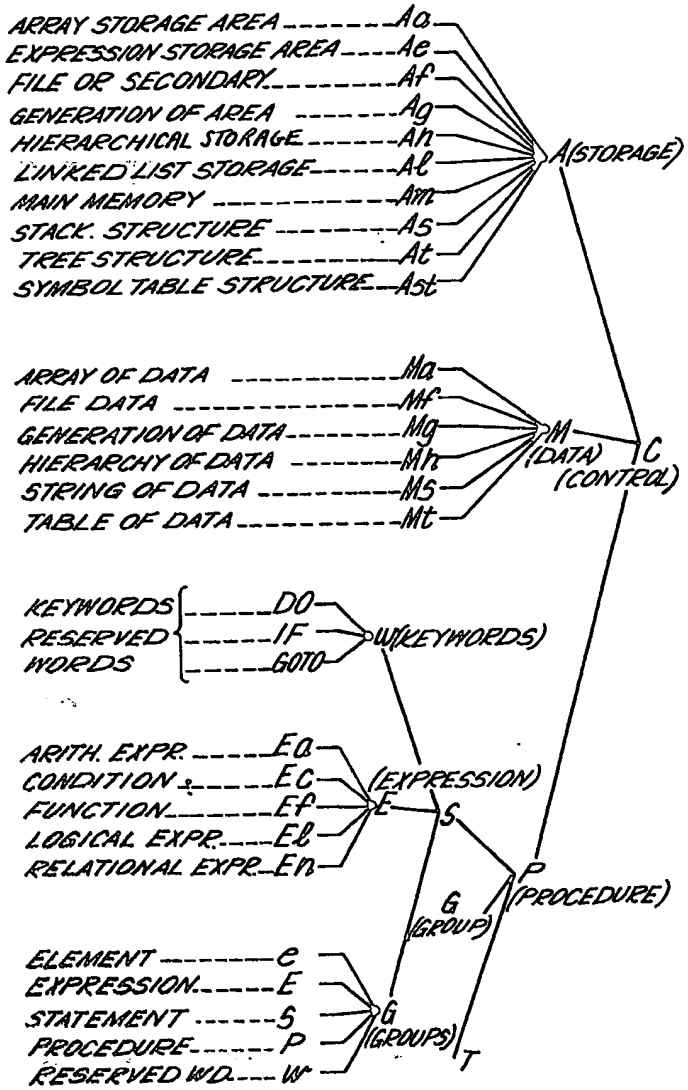


Fig.8. NEW METALANGUAGE MAJOR INFORMATION STRUCTURES

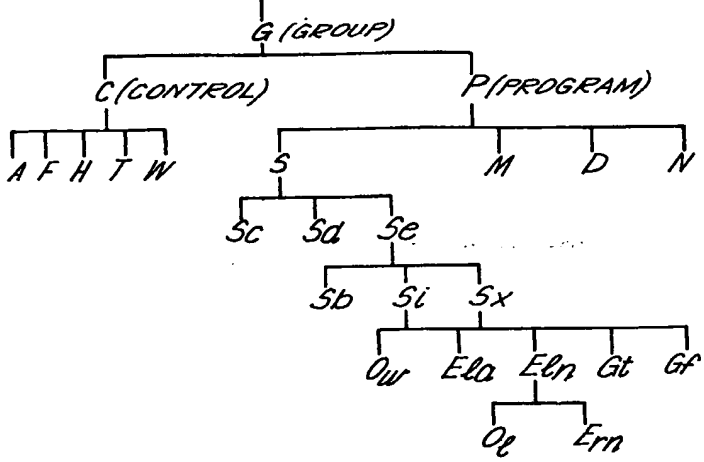
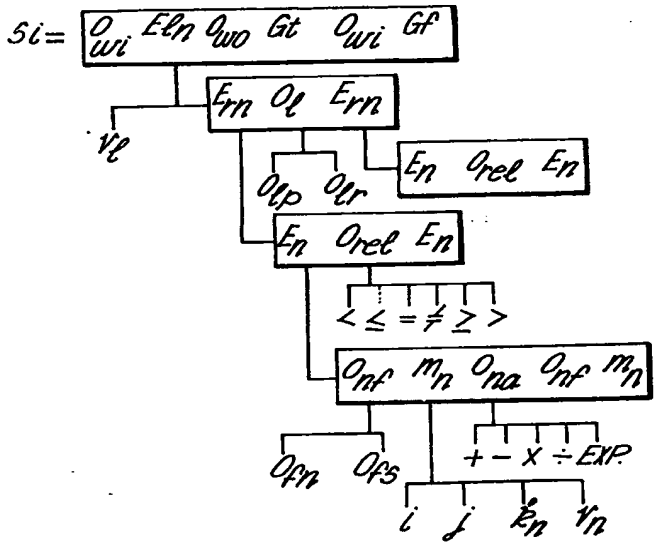


Fig.10. NEW METALANGUAGE "IF" STATEMENT



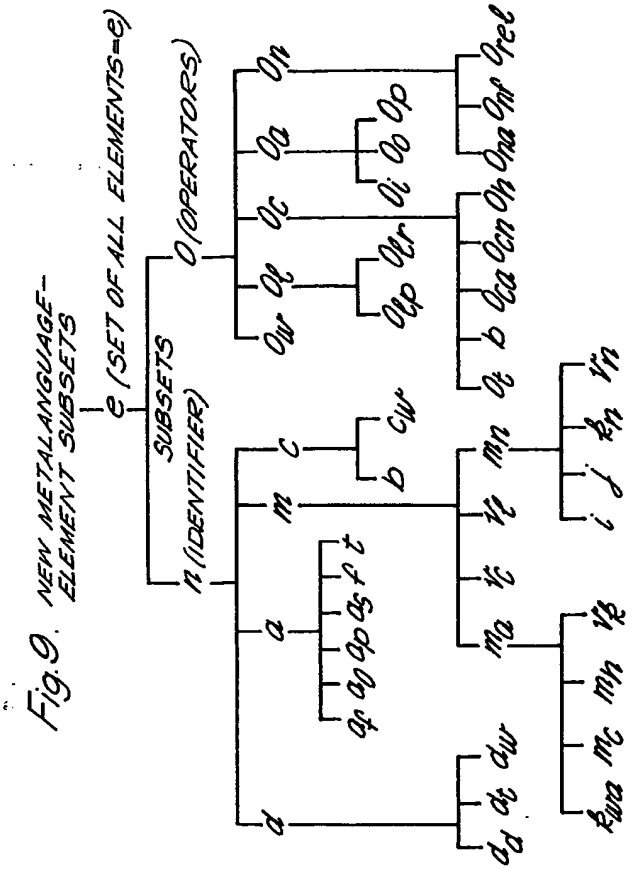

NEW METALANGUAGE-
ELEMENT SUBSETS
e (SET OF ALL ELEMENTS=e)
SUBSETS
O (OPERATORS)
n (IDENTIFIER)

Fig. 9.

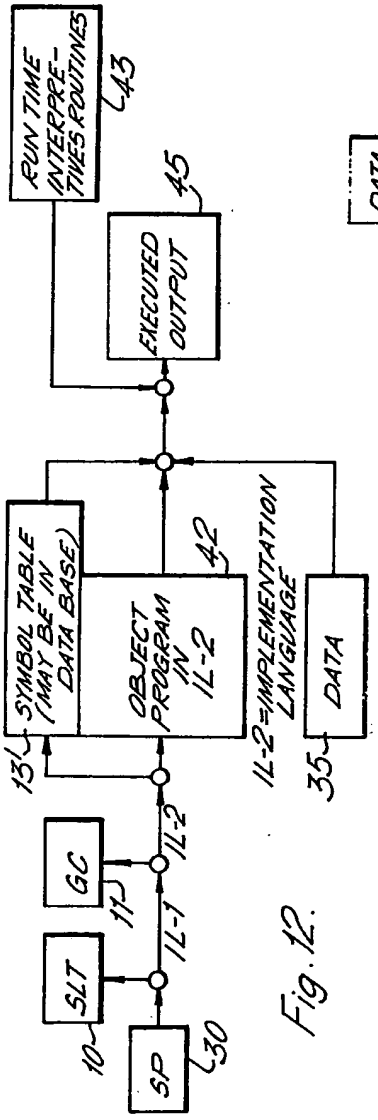

SP
30
SLT
10
GC
11
IL-1
IL-2
SYMBOL TABLE (MAY BE IN DATA BASE)
OBJECT PROGRAM IN IL-2
42
13
IL-2=IMPLEMENTATION LANGUAGE
DATA
35
RUN TIME INTERPRE-TIVES ROUTINES
43
EXECUTED OUTPUT
45

Fig. 12.

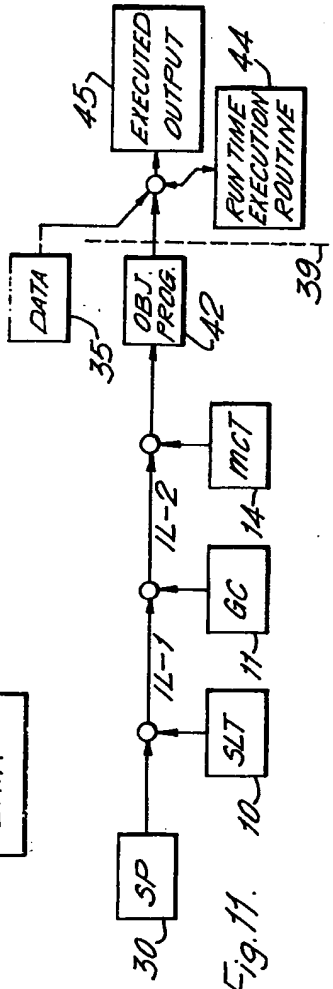

SP
30
SLT
10
GC
11
MCT
14
IL-1
IL-2
DATA
35
OBJ. PROG.
42
39
RUN TIME EXECUTION ROUTINE
44
EXECUTED OUTPUT
45

Fig. 11.

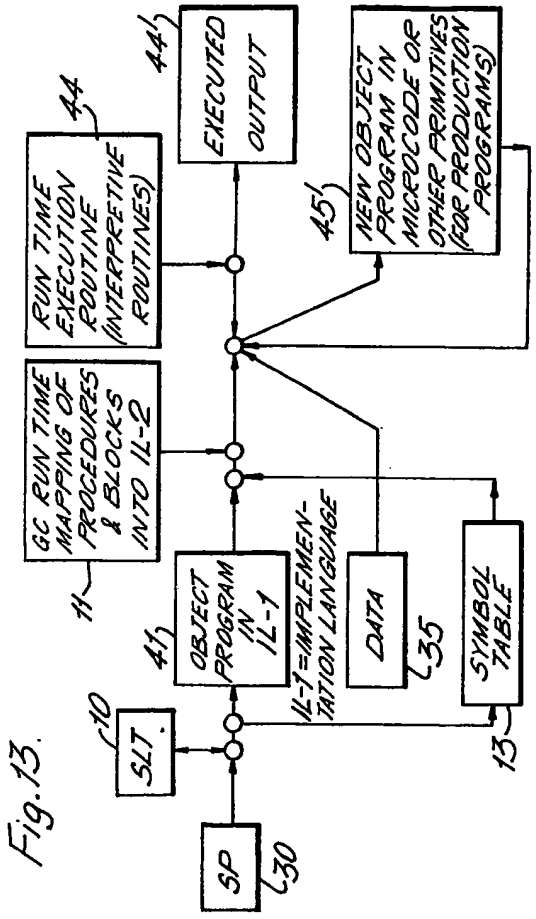

44
EXECUTED OUTPUT
RUN TIME EXECUTION ROUTINE (INTERPRETIVE ROUTINES)
45
NEW OBJECT PROGRAM IN MICROCODE OR OTHER PRIMITIVES (FOR PRODUCTION PROGRAMS)
GC RUN TIME MAPPING OF PROCEDURES & BLOCKS INTO IL-2
11
41
OBJECT PROGRAM IN IL-1
IL-1=IMPLEMENTATION LANGUAGE
DATA
35
SYMBOL TABLE
13
10
SLT.
SP
30

Fig. 13.

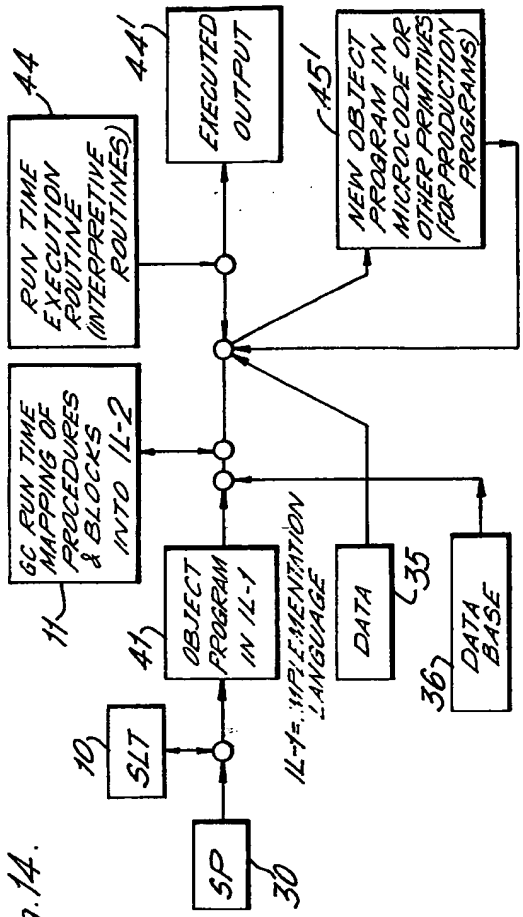

44
RUN TIME EXECUTION ROUTINE (INTERPRETIVE ROUTINES)
44'
EXECUTED OUTPUT
45'
NEW OBJECT PROGRAM IN MICROCODE OR OTHER PRIMITIVES (FOR PRODUCTION PROGRAMS)
GC RUN TIME MAPPING OF PROCEDURES & BLOCKS INTO IL-2
11
41
OBJECT PROGRAM IN IL-1
IL-1 = IMPLEMENTATION LANGUAGE
DATA
35
DATA BASE
36
10
SLT
SP
30

Fig. 14.